Bluetooth Mesh Networking. Aplicaciones y pruebas de concepto

Trabajo Fin de Máster

Curso 2019 - 2020

Realizado por

Lidia Fernández Fernández

Dirigido por

Francisco D. Igual Peña

Luis Piñuel Moreno

Máster en Internet de las Cosas

Facultad de Informática

Universidad Complutense de Madrid

Bluetooth Mesh Networking. Aplicaciones y pruebas de concepto

Bluetooth Mesh Networking. Applications and proofs of concept

---

Trabajo Fin de Máster en Internet de las Cosas

Departamento de Arquitectura de Computadores y Automática

Realizado por

Lidia Fernández Fernández

Dirigido por

Francisco D. Igual Peña

Luis Piñuel Moreno

Convocatoria: Septiembre 2020

Calificación: 9

Máster en Internet de las Cosas
Facultad de Informática
Universidad Complutense de Madrid

24 de Septiembre de 2020

Autorización de difusión

Lidia Fernández Fernández

Septiembre de 2020

La autora de este proyecto, matriculada en el Máster en Internet de las Cosas de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autora, el presente Trabajo Fin de Máster: *Bluetooth Mesh Networking. Aplicaciones y pruebas de concepto*, realizado durante el curso académico 2019-2020 bajo la dirección de Francisco Daniel Igual Peña y Luis Piñuel Moreno en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Dedicatoria

A los que nos dejaron en este 2020 y a todos esos valientes que, a pesar de todos los baches, siguen caminado.

Agradecimientos

En primer lugar, quiero agradecer a todos los profesores que durante mis años de formación han hecho que no pierda la ilusión por seguir aprendiendo. Gracias por haberme transmitido los conocimientos necesarios para tener a día de hoy más claro que elegí bien mi camino. Agradecer en especial a mi director Fran su ayuda durante todo este tiempo.

Este año sin duda no hubiese sido lo mismo sin ellos, así que, gracias a Richard, Francis, Adrián, Luis, Mathias, Allinson y Sergio por formar parte de esta experiencia, por haber compartido conmigo horas de clases, pero también por haber compartido risas fuera de ellas. Sabéis que aquí tenéis una amiga.

Agradecer también a mis amigos más cercanos y a mis familiares por estar ahí en los momentos de desconexión. Compartir horas de videollamadas con vosotros fue divertido, pero sin duda lo mejor fue volver a abrazarnos. Gracias por vuestros consejos y por animarme siempre.

Gracias a mis abuelos, pues tanto estando aquí abajo como estando allí arriba, he notado su apoyo y sus ganas de verme alcanzar mis metas. Pronto nos besaremos sin mascarilla.

Y, finalmente, gracias a mis padres y a mi hermana, pues son los pilares de mi vida, los primeros que celebran mis victorias y los primeros que me levantan en las derrotas. Gracias por la paciencia un año más y apoyarme de manera incondicional en cada una de mis decisiones. Sois el apoyo que toda persona desearía tener. Qué suerte la mía de teneros.

Resumen

Con el avance y el auge de Internet de las Cosas (IoT), las redes con topología en malla se han convertido en una solución popular, puesto que permiten la interconexión entre cientos, e incluso miles, de dispositivos IoT, los cuales han crecido de manera exponencial estos últimos años.

Hoy en día ya existen algunas tecnologías de comunicación inalámbrica exitosas que implementan dicha topología de red y que utilizan dispositivos de bajo coste y de bajo consumo. *Bluetooth Low Energy* ha sido históricamente una solución ampliamente aceptada dentro del mundo IoT debido a sus características. Con la incorporación de Bluetooth Mesh, que introduce la capacidad de desplegar topologías en malla en base a Bluetooth Low Energy, este protocolo se acerca a soluciones IoT a gran escala como, por ejemplo, automatización de edificios, redes de sensores, iluminación inteligente, etc.

Este documento pretende ser una guía para que el lector profundice en la tecnología Bluetooth Mesh, pues aparecen conceptos nuevos con respecto a Bluetooth LE, como, por ejemplo, proceso de aprovisionamiento, nodos, elementos, modelos, clave de aplicación, clave de red, etc.

Por otro lado, hay que tener en cuenta que el mercado ha evolucionado de la mano de esta novedosa tecnología, por lo que empresas líderes en el sector IoT han creado kits de desarrollo para implementar Bluetooth Mesh. En este caso, se hace uso de ESP-IDF, un framework desarrollado por la compañía *Espressif Systems*, que ayudará a implementar dos pruebas de concepto que muestren el funcionamiento de la malla Bluetooth a través de la configuración de múltiples modelos.

Palabras clave

Internet de las Cosas, topología en malla, Bluetooth Mesh, aprovisionamiento, modelo, ESP-IDF, ESP32, aplicaciones IoT.

Abstract

With the advancement and rise of the Internet of Things (IoT), mesh topology networks have become a popular solution, as they allow the interconnection between hundreds, and even thousands, of IoT devices, which have grown exponentially in recent years.

As of today, there are already some successful wireless communication technologies that implement such a network topology that use low-cost and low-power devices. *Bluetooth Low Energy* has historically been a widely accepted solution within the IoT world due to its characteristics. With the incorporation of Bluetooth Mesh, which introduces the ability to deploy mesh topologies based on Bluetooth Low Energy, this protocol approaches large-scale IoT solutions such as, for example, building automation, sensor networks, intelligent lighting, etc.

This document is intended to be a guide for the reader to delve into Bluetooth Mesh technology, as new concepts appear regarding Bluetooth LE, such as, for example, provisioning process, nodes, elements, models, application key, network key, etc.

On the other hand, it must be taken into account that the market has evolved hand in hand with this new technology, which is why leading companies in the IoT sector have created development kits to implement Bluetooth Mesh. In this case, use is made of ESP-IDF, a framework developed by the company *Espressif Systems*, which will help implement two proofs of concept that show the operation of the Bluetooth mesh through the configuration of multiple models.

Keywords

Internet of Things, mesh topology, Bluetooth Mesh, provisioning, model, ESP-IDF, ESP32, IoT applications.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En los últimos años, el Internet de las cosas (*Internet of Things,* IoT) se está volviendo más popular en la sociedad debido a su auge y avance tecnológico. Aunque la definición precisa de IoT a día de hoy es casi inalcanzable, se puede definir como el paradigma en el que las capacidades informáticas y de redes están integradas en cualquier tipo de objeto con el objetivo de usarlas para consultar el estado de dicho objeto y para cambiar su estado siempre que sea posible. También se podría definir de manera más simplificada como la capacidad de los dispositivos para comunicarse entre sí a través de, por ejemplo, una o más redes inalámbricas.

Con la expansión continua del Internet de las Cosas, se han empezado a utilizar cada vez más dispositivos conectados, desde teléfonos móviles hasta vehículos autónomos o sistemas de control. Esto provoca que la flexibilidad, agilidad y fiabilidad de las arquitecturas de red existentes sean cuestionadas, ya que se pone en duda si estas serán capaces de hacer frente a las crecientes demandas que les impone este crecimiento vertiginoso de las nuevas tecnologías.

En 2017, un informe predictivo de *Gartner*[1] [1] decía que el número de dispositivos IoT conectados en 2019 sería alrededor de los 14 mil millones, mientras que en 2021 se superarían los 25 mil millones. Tres años después, un informe de investigación de *Business Insider*[2] [2] ha confirmado que estas predicciones superaron a la realidad; sin embargo, eso no quita la sorprendente predicción actual, pues después de haber tratado con diferentes empresas y consumidores, se estima que habrá más de 41 mil millones de dispositivos IoT conectados para 2027, frente a los 8 mil millones de 2019.

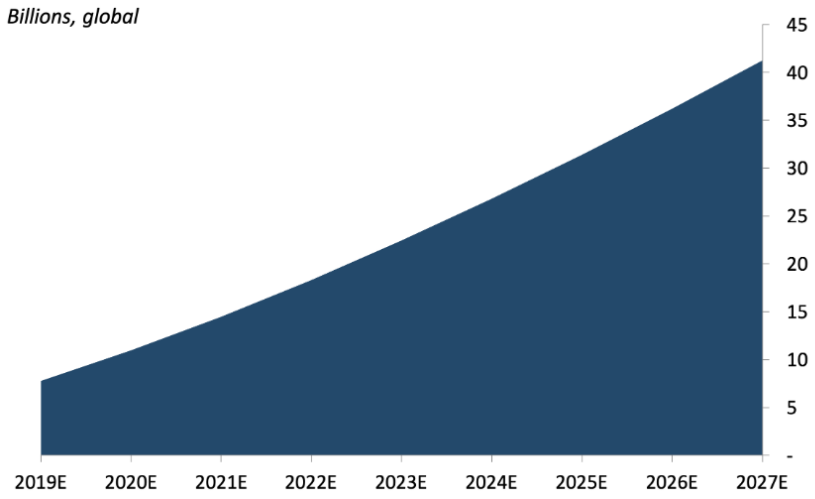

Figura 1. Predicción de dispositivos IoT conectados [2]

[1] *Gartner Inc.* es una empresa consultora y de investigación de las tecnologías de la información (*https://www.gartner.com/en*).
[2] *Business Insider* es un medio digital estadounidense de noticias financieras y empresariales, que también dedica publicaciones a la tecnología (*https://www.businessinsider.es/?r=US&IR=T*).

Según los datos ilustrados en la Figura 1, en los próximos años crecerá de manera exponencial el número de dispositivos IoT, por lo que es necesario buscar una opción atractiva de comunicación para cualquiera que desee utilizar en sus proyectos un amplio espectro de estos.

Las arquitecturas basadas en redes en malla (*mesh*) son una buena opción para dar cabida a las continuas oleadas de nuevos dispositivos IoT. Una red *mesh* no es más que una topología de red en la que cada nodo perteneciente a ella está conectado a todos los demás nodos de la red. Este tipo de topología presenta una gran cantidad de beneficios, entre ellos, la *confiabilidad* en escenarios donde la conectividad es primordial, pues si alguno de los nodos falla, no se dificulta la transferencia de datos debido al encaminamiento automático de estos. Además, cuanto más grandes sean los despliegues, menos problemas habrá con la distancia de cobertura de red, pues se ampliará al permitir extender el alcance de la misma.

A pesar de contar con estas ventajas, en la industria no existe únicamente un protocolo de comunicación inalámbrica que adopte esta topología de red, por lo que resulta necesario determinar cuál es el más apropiado en base a las condiciones y expectativas del proyecto a desarrollar.

1.1. Motivación

En los últimos años, cada vez es más común buscar satisfacer en el área de las tecnologías de comunicación requisitos como la *interoperabilidad*, la posibilidad de dar cobertura en *áreas grandes*, reducción y optimización del *consumo energético*, capacidad de controlar y monitorear una gran *cantidad de dispositivos*, etc.

Dentro de las comunicaciones inalámbricas, las que adoptan las topologías en malla (*mesh*) son las indicadas para satisfacer esa serie de requisitos. Actualmente hay una amplia variedad de estas, pero algunas presentan problemas relacionados con las bajas velocidades de transmisión de datos, el número limitado de "saltos" al transmitir datos a través de la malla, límites de escalabilidad y dificultades al seguir los procedimientos para cambiar la composición del dispositivo de la red en malla.

La creación de una tecnología de red en malla estándar basada en *Bluetooth Low Energy* (BLE) [3] brindó a los desarrolladores de aplicaciones IoT la oportunidad de cumplir con los requisitos solicitados, pero sin las limitaciones y restricciones asociadas, ya que, después de todo, la interoperabilidad y la eficiencia energética son las características distintivas de Bluetooth LE.

Con la llegada de Bluetooth Mesh, no es necesario agregar ningún hardware adicional, ya que se puede aplicar a cualquier dispositivo BLE cuya versión sea la 4.0 o superior. Esto le hace reunir más puntos a favor, pues como se sabe, Bluetooth LE presenta gran compatibilidad con teléfono móviles, ordenadores, tabletas y otro tipo de periféricos, como, por ejemplo, las pulseras inteligentes, lo cual satisface otro de los requisitos.

A diferencia de Bluetooth BR/EDR, que solo admitía conexión punto a punto (*1:1*) o Bluetooth Low Energy, que además de admitir la comunicación punto a punto, tenía la

capacidad de difusión de datos (*1:m*), Bluetooth Mesh permite la comunicación de "muchos a muchos" (*m:m*), lo que es ideal para aplicaciones IoT como la automatización de edificios, la iluminación comercial, el seguimiento de activos o las redes de sensores, que requieren cientos o miles de dispositivos para comunicarse de manera fiable y segura.

1.2. Objetivos

El objetivo principal de este proyecto será doble. En primer lugar, se describirá de manera detallada Bluetooth Mesh, pues tras estudiar toda la documentación que existe sobre esta tecnología, se logrará realizar una guía que hable tanto de su funcionamiento e implementación como de la seguridad que le rodea. En segundo lugar, se comprobará que el *framework* ESP-IDF [3] da soporte a la malla Bluetooth. Para hacer esto posible, se realizarán una serie de desarrollos *software* con algunos casos de uso puntuales sobre el SoC ESP32 [4].

Para poder lograr estos objetivos principales, es necesario definir unos objetivos secundarios, los cuales aportarán una base sólida de conocimientos para el desarrollo del trabajo. La lista de estos objetivos será:

- Analizar y documentar el estado actual de tecnologías de comunicaciones inalámbricas que implementan la topología de red en malla.
- Documentar la parte de la pila de Bluetooth Low Energy sobre la que se asienta Bluetooth Mesh.
- Estudiar las diferentes herramientas de desarrollo que se proporcionan en el mercado y que dan soporte a la tecnología BLE Mesh.
- Recoger información sobre el SoC seleccionado y la placa de desarrollo para después documentar de manera clara y concisa.
- Analizar, estudiar y documentar los diferentes mecanismos de aprovisionamiento de dispositivos Bluetooth LE.
- Analizar aplicaciones IoT en las que se use Bluetooth Mesh. Dar algún ejemplo real de estas.
- Estudiar algunos de los diferentes modelos que propone la comunidad Bluetooth para comenzar con el desarrollo de aplicaciones.
- Desarrollo software de pruebas de concepto dedicadas a los modelos específicos de Bluetooth Mesh.
- Desarrollo de un panel de control haciendo uso de la herramienta Node-RED.
- Comprobar el correcto funcionamiento de los sistemas implementados mediante la simulación de distintos escenarios.

1.3. Metodología

Se entiende por metodología, al conjunto de procedimientos que nos permiten alcanzar un objetivo determinado. En este caso, se quiere alcanzar la lista de objetivos mencionados anteriormente. Para ello, el trabajo se ha dividido en tres etapas con el fin de conseguir el mayor éxito posible en el desarrollo de este.

i. Etapa de investigación:

Esta primera etapa se dividirá en dos, pues se realizarán dos investigaciones diferentes. Inicialmente, se comenzará analizando el estado actual de las diferentes tecnologías de comunicación inalámbrica que admiten la topología de red en malla. De esta manera, se creará una idea fundamentada de los pros y contras que presentan cada una de ellas, y, además, se dará pie a introducir la novedosa tecnología sobre la que gira este proyecto, es decir, Bluetooth Mesh. La segunda investigación estará ligada a las distintas empresas que aseguran tener kits de desarrollo que soportan BLE Mesh. A través de ella, se analizarán los diferentes recursos que ofrecen, sin embargo, estará de la mano del propio desarrollador de soluciones Bluetooth Mesh seleccionar, por ejemplo, la plataforma *hardware* con la que desee trabajar.

ii. Etapa de estudio:

Al igual que la etapa anterior, el estudio del trabajo tomará dos caminos diferentes. Primero se debe conseguir describir en detalle Bluetooth Mesh, por lo que será necesario contar con toda la documentación oficial. Tras el estudio de esta, se podrán añadir al presente documento nuevos conceptos con los que progresivamente se irá consiguiendo una idea general de esta tecnología. Entre estos conceptos se encontrarán el término aprovisionamiento (*provisioning*), el tipo de modelo de comunicación que se usa entre los nodos, la seguridad basada en la separación de conceptos, etc. Después, el estudio estará centrado sobre el entorno de desarrollo sobre el que se va a implementar una solución BLE Mesh. Para ello, habrá que seguir las pertinentes guías de instalación, configuración y programación en ESP-IDF.

iii. Etapa de desarrollo:

En esta última etapa se pretende comprobar que el entorno de desarrollo ESP-IDF proporciona el soporte adecuado para que los desarrolladores puedan realizar con éxito soluciones Bluetooth Mesh en aplicaciones IoT. Para ello, se crearán diferentes pruebas de concepto donde se trabajarán con distintos modelos de esta tecnología de red. Se explorarán aspectos como la programación desde cero de un modelo básico para después comprobar cómo se aprovechan las ventajas de una topología en malla, como, por ejemplo, el aumento del área de cobertura a pesar de estar tratando con dispositivos Bluetooth LE cuyo rango de cobertura es más reducido. De igual modo, se desarrollará una prueba de concepto más elaborada en la que coexistirán varias tecnologías para obtener una solución completa de sensorización.

1.4. Estructura de la memoria

El presente documento se compone de nueve capítulos, los cuales se describen a continuación de manera breve.

- El primer capítulo sirve a modo de *introducción*, pues además de presentarse el proyecto, se incluye la motivación, se plantean los objetivos y se describe la metodología a seguir para el desarrollo de este.

- En el segundo capítulo se introduce el *estado del arte*, en el que se detallan las actuales tecnologías de comunicaciones inalámbricas que admiten la topología en malla en sus implementaciones.

- El tercer capítulo es la base del proyecto en sí. En él se describe en detalle Bluetooth Mesh con el objetivo de que el lector pueda tener los conocimientos necesarios antes de proceder a los siguientes puntos.

- En el cuarto capítulo se describen las herramientas seleccionadas tanto *hardware* como *software* para el desarrollo del proyecto.

- En el quinto capítulo se hablan de posibles *aplicaciones* IoT en las que se pueda usar Bluetooth Mesh, así como soluciones ya desarrolladas y que han tenido éxito.

- En el capítulo sexto se documentarán las *pruebas de concepto* realizadas, las cuales ayudan a comprobar si ESP-IDF da el soporte a Bluetooth Mesh, y, por tanto, se consiguen unos resultados exitosos.

- En el séptimo capítulo se presentan las *conclusiones* del proyecto y algunas ideas de futuro que pueden aportar valor al trabajo.

- En el octavo y noveno capítulo se encuentran los capítulos de la *introducción* y las *conclusiones* traducidos al inglés.

- Finalmente, mencionar que, tras la bibliografía, existen una serie de apéndices muy relevantes que completan la lectura del trabajo. En el primero de ellos, se tratan las primeras capas de la pila del protocolo Bluetooth Low Energy, mientras que, el segundo apéndice es una guía de instalación y configuración del entorno de desarrollo ESP-IDF. Por último, el tercer apéndice ofrece tres soluciones reales de aprovisionamiento.

Capítulo 2

Estado del arte

Cuando se trata de conectividad inalámbrica, un desarrollador de soluciones IoT dispone de un gran abanico de opciones para elegir. Sin embargo, en ocasiones, no se tiene clara cuál es la tecnología adecuada, ya que la elección final de una tecnología (o tecnologías) de conectividad para un proyecto se reduce en los requisitos y las necesidades de ese proyecto específico.

El presente trabajo está enfocado en las redes en malla, por lo que a continuación, se describirán las actuales tecnologías de comunicación inalámbrica que adoptan esta topología de red.

2.1. Tecnologías de red en malla

Antes de tratar cada una de las tecnologías, es importante hablar de los atributos [7] o propiedades que se deben tener en cuenta a la hora de comparar cada una de ellas.

En primer lugar, es necesario conocer el *rango*, es decir, la distancia máxima a la que pueden lograr una conexión confiable entre los dispositivos de un sistema. En segundo lugar, sería interesante conocer el *rendimiento o ancho de banda*, que es la velocidad máxima de datos admitida por las conexiones de los dispositivos. También habría que considerar el atributo de la *movilidad*, pues sería bueno saber si los dispositivos deben ser físicamente fijos o móviles. Otro atributo sería el del *consumo de energía* que necesita el dispositivo en cuestión, y, por último, habría que tratar *coste*, tanto del dispositivo que implementa la tecnología como la instalación de esta. En base a estos parámetros, se realiza a continuación un estudio de las principales tecnologías de comunicación inalámbrica con soporte para *mesh* disponibles a día de hoy.

2.1.1. Z-Wave

Z-Wave [9] comenzó como protocolo para controlar sistemas de iluminación, pero evolucionó hasta convertirse en un protocolo de automatización del hogar administrado por la *Z-Wave Alliance*[3]. Actualmente, su uso principal reside en aplicaciones de hogares inteligentes (*Smart Home*).

Esta tecnología es compatible con redes con topología en malla, por lo que no necesita un nodo que coordine las comunicaciones. Además, es escalable, pues permite controlar hasta 232 dispositivos, con una duración de la batería de estos de hasta dos años.

Z-Wave opera en la banda de 908/915 MHz en EEUU, mientras que en Europa opera en la banda de 868 MHz. Está diseñada de este modo para evitar interferencias con la banda ISM

[3] https://z-wavealliance.org/

de 2.4 GHz. Además, alcanza velocidades de datos de hasta 100 kbps y tiene un rango de cobertura entre 30 y 50 metros en interior.

La ventaja que presenta Z-Wave es que al trabajar con él se encuentran menos interferencias, pues funciona fuera de la banda estándar que ya está inundada por otras tecnologías como Zigbee, Wi-Fi o Bluetooth, como se verá más adelante. Por otro lado, una de sus desventajas es la interoperabilidad, pues no se garantiza que funcione en todos los países debido a las diferentes frecuencias de operación. Además, el uso de esta tecnología suele ser más costoso que las alternativas.

2.1.2. Zigbee

El protocolo Zigbee [10] fue creado por las empresas miembro de *Zigbee Alliance*[4], y se usa más comúnmente en aplicaciones de baja velocidad de datos que requieren una alta escalabilidad. Actualmente, su uso se centra en el monitoreo y control inalámbrico dentro del espacio del hogar inteligente o *wearables*.

La tecnología Zigbee está construida sobre la especificación IEEE 802.15.4[5], la cual se utiliza principalmente para transmitir pequeñas cantidades de datos (250 kbps) mientras se mantiene un bajo consumo de energía. Opera en la banda de 2.4 GHz y tiene un alcance de hasta 100 metros.

Si bien se admite una topología en estrella, la topología más utilizada por Zigbee es la topología de malla, puesto que puede manejar potencialmente más de 65000 nodos en una red mientras mantiene la fuerza de la señal y la plena capacidad de enviar y recibir datos.

Gracias al uso de Zigbee se puede contar con ventajas como la escalabilidad, pues como se acaba de comentar, permite un número muy alto de nodos. Asimismo, destaca por su bajo consumo de energía y su bajo coste. Sin embargo, no destaca por su seguridad, pues es vulnerable debido a que todos los fabricantes no siguen estrictamente los estándares, por lo que un solo dispositivo vulnerable puede comprometer todo el sistema.

2.1.3. 6LowPAN

La tecnología 6LowPAN (*IPv6 Low-Power Wireless Personal Area Network*) [11] da soporte a la comunicación IPv6[6] sobre la tecnología inalámbrica IEEE 802.15.4, y actúa como una capa de adaptación entre la tecnología estándar IPv6 y el medio ofrecido por IEEE 802.15.4 para comunicaciones con pérdidas y de bajo consumo.

Al introducir la pila IPv6, se da un avance en el ecosistema IoT, pues aumenta el rango de direccionamiento, permitiendo así que cualquier dispositivo tenga su propia dirección IP para conectarse a Internet. Además, una de las topologías que soporta es en malla, lo cual le hace crear redes robustas y escalables.

---

[4] https://zigbeealliance.org/

[5] https://standards.ieee.org/standard/802_15_4-2020.html

[6] IPv6 (*Internet Protocol versión 6*) fue diseñado como la solución a la necesidad de interconexión e identificación masiva de dispositivos en Internet. IPv6 proporciona un espacio de direccionamiento mayor a IPv4, principal razón de su introducción y éxito.

Sus principales aplicaciones IoT pueden ser en domótica, campos de cultivo o automatización de edificios, ya que 6LoWPAN se caracteriza por su bajo consumo energético y bajo coste, pero cuenta con una velocidad de transferencia en torno a los 250 kbps.

Por último, mencionar que, para implementar este protocolo se debe de usar un hardware específico para dar soporte a 802.15.4.

2.1.4. Thread

El protocolo de red Thread [12] es desarrollado, actualizado y gestionado por *Thread Group*[7]. Está basado en 6LowPAN y fue diseñado como una solución de red escalable, de bajo consumo, confiable y segura para conectar dispositivos IoT. Se puede usar como complemento Wi-Fi, sobre todo en el ámbito de las configuraciones domóticas, debido a los problemas de cobertura. Por tanto, se podría decir que dentro de las aplicaciones IoT, Thread destaca por su uso en automatización de edificios y *Smart Homes.*

Esta tecnología es compatible con redes *mesh*, por lo que es capaz de manejar hasta 250 nodos con altos niveles de autenticación y cifrado. Además, Thread está diseñado para trabajar sobre dispositivos compatibles con IEEE 802.15.4.

2.1.5. LoRaWAN

LoRaWAN [13] es un protocolo de red inalámbrica mantenido por *LoRa Alliance*[8] que implementa redes de área amplia (WAN) con características específicas, como el bajo consumo, que le hacen popular para soportar comunicaciones seguras en aplicaciones IoT como *Smart Cities*, *Smart Parking*, seguimiento de activos en cadenas de suministros y logística, etc.

LoRaWAN se basa en LoRa, pues es uno de los varios protocolos que definen las capas superiores de una red LoRa. Permite el enrutamiento, la gestión de las frecuencias de comunicación del dispositivo, las velocidades de datos y la potencia de transmisión. LoRa, por su parte, es un protocolo de capa física en comunicaciones inalámbricas patentado y desarrollado por la empresa *Semtech*[9] para trabajar en bandas alrededor de 900 MHz. Por ejemplo, en la Unión Europea, opera en la banda de los 868 MHz.

A diferencia de los protocolos vistos hasta ahora, LoRaWAN tiene un alcance superior, pues es capaz de alcanzar los 5 Km en entornos urbanos y los 15 Km en entornos rurales. Sin embargo, no destaca por sus velocidades de transferencia de datos, pues estas van desde los 0.3 kbps hasta los 50 kbps.

Normalmente, LoRaWAN implementa una topología en estrella, pero también es posible construir redes en malla para cubrir un área mayor y aumentar la seguridad de la red. En esta tecnología, la topología en malla se crea en tiempo real y el mapa de enrutamiento

[7] https://www.threadgroup.org/
[8] https://lora-alliance.org/
[9] https://www.semtech.com/

óptimo se actualiza periódicamente. Por tanto, si un nodo de la red funcionara mal, otro nodo retomaría esas tareas y así se evitaría la interrupción de la red.

Por último, y a modo de curiosidad, *Pycom LoPy4*[10] es una placa de desarrollo con soporte para múltiples tecnologías inalámbricas (LoRa, Wi-Fi, BLE, etc.). *Pycom* permite desarrollar ejemplos reales de topologías punto-a-punto o en estrella, pero también es capaz de implementar topología en malla, puesto que se basa en la especificación *OpenThread*[11]. Por tanto, la tecnología LoRa, gracias a esta placa, también tiene soporte *mesh* (LoRaMESH) [14], pues con ella se pueden hacer configuraciones de una red LoRaMESH, se pueden descubrir vecinos y enviar paquetes a todos ellos, etc.

2.1.6. Wi-Fi

Wi-Fi es el nombre comercial de cualquier red de área local inalámbrica (WLAN) que sigue el estándar IEEE 802.11[12]. Hay muchas variantes de Wi-Fi, pero *Wi-Fi Alliance*[13] ha adoptado recientemente un sistema de numeración de versiones, las cuales apuntan a diferentes tipos de aplicaciones, incluidas las que están pensadas para un mayor alcance, un mayor rendimiento o una mejor cobertura. En el caso de las aplicaciones IoT, Wi-Fi se usa más comúnmente para dispositivos que necesitan una conexión directa a Internet. Sin embargo, no suele asociarse a las que requieren un bajo consumo de energía, pues su uso está limitado en aplicaciones y dispositivos que requieren funcionar con baterías pequeñas durante largos períodos de tiempo.

Wi-Fi opera en las bandas ISM de 2,4 y 5 GHz, y su velocidad de transferencia llega hasta los 600 Mbps, aunque lo habitual es entre los 150 y 200 Mbps. Además, su rango de cobertura es de aproximadamente 50 metros.

La topología más popular utilizada en Wi-Fi es la topología en estrella, en la que los nodos solo pueden comunicarse entre sí a través de un *hub* central; sin embargo, también admite la topología de red en malla, lo cual permite solucionar los problemas de conectividad sin ser necesarios los repetidores Wi-Fi que se usaban antes. La principal diferencia entre una red Wi-Fi *mesh* [15] y una red Wi-Fi construida con repetidores es que en el primer caso se tienen nodos que se conectan entre sí, por lo que los diferentes puntos de acceso no sólo están conectados al router, sino que también son capaces de conectarse entre ellos, lo que permite una mejor cobertura y una mejor gestión de la red.

Por tanto, con Wi-Fi *mesh* se podrían escalar hasta 1000 nodos en grandes áreas sin requerir ningún soporte de infraestructura Wi-Fi específico. Dentro de las posibles aplicaciones IoT con esta tecnología de comunicación inalámbrica estaría la iluminación inteligente, el hogar inteligente y la automatización de, por ejemplo, grandes aparcamientos.

Las principales ventajas de este protocolo es que está integrado en dispositivos del día a día, como son los teléfonos, los ordenadores, las tablets, etc., y, además, permite unas velocidades de transmisión muy altas. Sin embargo, algunas desventajas serían su alto

[10] https://pycom.io/product/lopy4/
[11] https://openthread.io/guides/thread-primer
[12] https://standards.ieee.org/standard/802_11-2016.html
[13] https://www.wi-fi.org/

consumo de energía y que puede ser *hackeable*, ya que las redes Wi-Fi a menudo están conectadas directamente a Internet, lo que crea vulnerabilidades para el acceso remoto malicioso.

2.1.7. Bluetooth

La tecnología Bluetooth es una solución inalámbrica de bajo consumo que funciona en la banda ISM de 2,4 GHz. Se ha expandido a lo largo de los años y ahora ofrece una gran flexibilidad en topologías de red, ancho de banda y comunicaciones para abordar diferentes aplicaciones IoT.

Actualmente se cuenta con dos opciones de radio Bluetooth. Por un lado, está la especificación clásica de Bluetooth (BR/EDR) y por otro lado está Bluetooth Low Energy o Bluetooth Smart. Este último se caracteriza por ser un protocolo importante para desarrollar aplicaciones IoT, pues además de contar con un consumo de energía significativamente reducido, puede alcanzar hasta una cobertura de entre 50 y 150 metros y una velocidad de transferencia de datos de hasta 2 Mbps.

Bluetooth LE brinda la opción de operar en topologías punto a punto, en estrella y en malla. Esta última topología mencionada, que surgió en 2017 [16], permite que los nodos se conecten directamente entre sí sin necesidad de comunicarse con otros a través de un *hub* central.

Bluetooth Mesh, que es como se conoce a esta tecnología de red, además de permitir crear sistemas seguros y confiables con más de 32000 dispositivos, aprovecha la presencia de BLE en los teléfonos, tablets, PCs y otros dispositivos inteligentes para tomar ventaja dentro de las soluciones IoT, pues cuenta con el hardware BLE ya existente. Sin embargo, uno de los riesgos a los que un desarrollador podría enfrentarse sería que al ser una tecnología emergente su configuración es algo compleja, por lo que se requiere una cierta comprensión de sus funcionalidades.

2.2. Comparativa de soluciones

Tras haber visto en el punto anterior siete tecnologías de comunicación inalámbrica que adoptan la topología *mesh*, se debe evaluar cuál de ellas es más adecuada para llevar a cabo una aplicación en concreto. En la Tabla 1 se muestra una comparativa [8] de las diferentes soluciones.

	Z-Wave	Zigbee	6LowPAN	Thread	LoRaWAN	Wi-Fi	Bluetooth
Rango	30-50 m	30-100 m	-	30-100 m	2-20 Km	15-100 m	10-150 m
Rendimiento	10 kbps – 100 kbps	20 kbps – 250 kpbs	-	20 kbps – 250 kpbs	0.3 kbps – 50 kbps	54 Mbps – 1.3 Gbps	125 kbps – 2 Mbps
Energía consumida	Bajo	Bajo	Bajo	Bajo	Bajo	Medio	Bajo
Frecuencias	900 MHz	2.4 GHz	2.4 GHz	2.4 GHz	Varias	2.4, 5 GHz	2.4 GHz
Nº máx nodos	232	65000	-	250-300	-	-	32000
Duración batería	Hasta 2 años	6 meses a 2 años	1 a 2 años	6 a 10 años	7 a 12 años	días	Hasta 2 años
Coste	Moderado	Moderado	Bajo	Moderado	Moderado	Moderado	Bajo
Topología	Mesh	Star, Mesh	Star, Mesh	Mesh	Star, Mesh	Star, Mesh	P2P, Broadcast, Star, Mesh

Tabla 1. Comparación tecnologías

Aunque existen diferencias, la mayoría de estos protocolos comparten un objetivo común, y es el de hacer que el consumo de energía de la comunicación inalámbrica sea lo más bajo posible para extender la vida útil de los dispositivos IoT. No obstante, el objetivo de este apartado no es el de decir qué tecnología es mejor, porque cada una de ellas cuenta con sus pros y contras, sino de facilitar al desarrollador de soluciones IoT a través de la Tabla 1, cuándo es mejor usar una u otra tecnología en el momento en que se tenga definido el proyecto en el que se desee implementar, pues en este, llegado un determinado momento, se deberá evaluar qué tecnología es más adecuada para una aplicación en concreto.

Es cierto que hay casos de uso específico donde la implementación de una tecnología es obvia. Por ejemplo, para aplicaciones que requieren un gran ancho de banda y a su vez se necesitan grandes transferencias de datos, es lógico pensar que el uso de Wi-Fi tiene más sentido que el resto de las tecnologías enumeradas. Por otro lado, si la aplicación implica implementaciones de bajo ancho de banda y largo alcance, como en exteriores y áreas rurales, entonces tecnologías como LoRaWAN tienen más sentido.

Obviamente, también se puede dar el caso en el que para una determinada aplicación se puede elegir entre varias de las tecnologías propuestas. Por ejemplo, el mercado de *Smart Home* está liderado por productos basados en Z-Wave y Zigbee, aunque también existen productos Thread y Wi-Fi. Sin embargo, la nueva especificación de Bluetooth, es decir, Bluetooth Mesh, pretende ser otro competidor en este sector IoT, ya que brinda mayor flexibilidad y buena balanza entre consumo de energía, confiabilidad, latencia y rendimiento.

Por otra parte, y antes de pasar a la siguiente sección, si se quiere seguir con la comparativa, es importante saber que las redes en malla se caracterizan por retransmitir datos utilizando la técnica de encaminamiento por inundación (*flooding*) o la técnica de encaminamiento por enrutamiento (*routing*). La mayoría de los protocolos vistos usan la técnica de enrutamiento, en la que los mensajes se envían a través de los nodos por el camino establecido más apropiado entre origen y destino, actualizando en todo momento las tablas de enrutamiento. Por el contrario, Bluetooth Mesh utiliza el enfoque de inundación [4], en

el que cada paquete no es enviado a lo largo de una ruta específica que comprende una secuencia de solo ciertos dispositivos, sino que todos los dispositivos reciben todos los mensajes de los nodos que están en rango directo y, en caso de que sean nodos de retransmisión (*relay*), el mensaje se retransmitirá nuevamente para llegar a los nodos que están más alejados del nodo origen. En términos generales, la técnica de *flooding* tiene sus pros y contras; sin embargo, en Bluetooth Mesh se explotan más sus ventajas, pues utiliza el enfoque de inundación administrada o gestionada, que implementa una limitación de tiempo de vida (*TTL*) para controlar el número máximo de "saltos" sobre los que se retransmite un mensaje. Además, todos los nodos implementan lo que se conoce como caché de mensajes, que contiene todos los mensajes vistos recientemente. En el momento en el que un nodo encuentra un mensaje en la caché, considera haberlo procesado antes y se descarta de manera inmediata. Por tanto, el uso de la inundación administrada permite que los mensajes lleguen a su destino a través de múltiples rutas, lo cual mejora la confiabilidad de la red potencialmente porque si algún nodo cae, la red no se inutiliza, pues hay múltiples rutas por las cuales el mensaje puede llegar a su destino.

Para este trabajo final del máster en Internet de las Cosas, la tecnología *mesh* seleccionada será Bluetooth, pues lo que se busca con este proyecto es explorar esta novedosa tecnología, que, como se verá más adelante en más detalle, se caracteriza principalmente por su fiabilidad, escalabilidad, seguridad e interoperabilidad.

Capítulo 3

Bluetooth Mesh

En 2017, Bluetooth SIG[14] lanzó el protocolo Bluetooth Mesh como su última tecnología desarrollada [17]. Se trata de una tecnología de red, no de una tecnología de comunicación inalámbrica, puesto que la pila Bluetooth Mesh está construida sobre la pila existente de Bluetooth Low Energy (véase Apéndice I).

Una red de Bluetooth Mesh soporta una topología de comunicación *many-to-many*, y consiste en una colección de dispositivos BLE especialmente preparados (en adelante, nodos). Puede haber miles de nodos en una misma red en malla y que estos se comunican entre sí mediante el envío de mensajes. Al tratar con esta topología de comunicación, no hace falta que todos los nodos estén en rango directo, ya que los mensajes pueden saltar (*multi-hop*) de un nodo a otro hasta llegar a su destino. Además, para hacer BLE Mesh más fiable, se hacen copias de los mensajes para que estos lleguen a su destino a través de los múltiples caminos (*multi-path*).

3.1. Pila del protocolo Bluetooth Mesh

Como bien se ha mencionado, este nuevo protocolo está construido sobre la pila de BLE. Esta última pila se dividía en los bloques de *Application, Host* y *Controller*. En este caso, el bloque *Controller*, que contenía capas como la física y la de enlace, se mantiene, puesto que proporciona las capacidades fundamentales de las comunicaciones inalámbricas, mientras que el bloque *Host* es sustituido por un nuevo bloque.

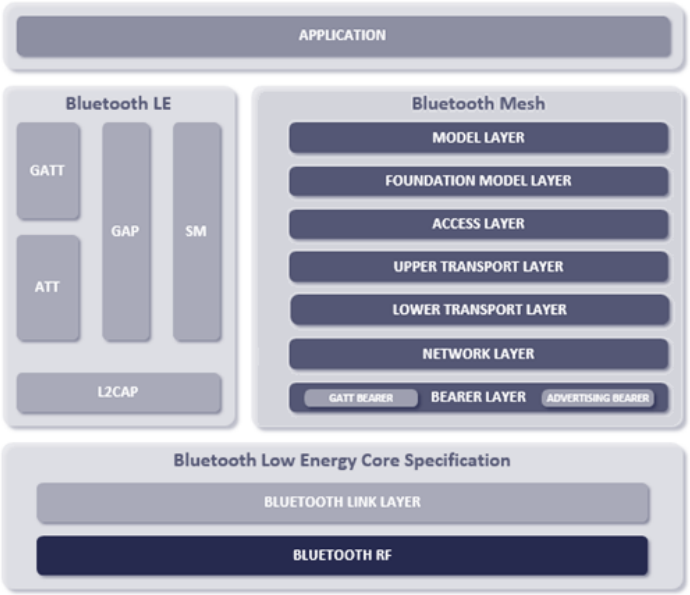

Figura 2. Pila de los protocolos BLE y BLE Mesh [19]

[14] Bluetooth Special Interest Group (*https://www.bluetooth.com/*) es una organización sin ánimo de lucro que define los estándares de Bluetooth y continúa impulsando el desarrollo de esta tecnología inalámbrica. Los promotores más importantes del SIG son: Ericsson, Nokia, Intel, Microsoft, Apple, Toshiba y Lenovo.

La pila de Bluetooth Mesh contiene 7 capas implementadas sobre la pila del protocolo Bluetooth Low Energy (Figura 2). A continuación, se explica cada capa de manera detallada.

- *Bearer Layer*:

  Esta capa, además de definir cómo se intercambian los mensajes por los diferentes nodos de la red, sirve como interfaz para conectar las capas superiores de la pila Bluetooth Mesh y la especificación Bluetooth Low Energy. Por tanto, debido a la compatibilidad que proporciona la capa *bearer*, se puede usar Bluetooth Mesh a través del hardware Bluetooth.

  Los dos *bearers* definidos son *advertising bearer* y *GATT bearer*. El portador de publicidad o *advertising bearer,* es el portador principal. Aprovecha las funciones de publicidad y escaneo GAP[15] de Bluetooth LE para transmitir y recibir las PDUs (*protocol data unit*). Por otro lado, el portador GATT o *GATT bearer,* permite que un dispositivo que no es compatible con el *advertising bearer* se comunique indirectamente con los nodos de una red de malla que sí lo hacen, es decir, los nodos proxy. Estos nodos son compatibles con el *GATT bearer* y con el *advertising bearer*, pues permite convertir y retransmitir mensajes entre los dos tipos de portador.

- *Network Layer*:

  La capa de red se encarga principalmente de definir cómo se dirigen los diversos tipos de mensajes hacia uno o más elementos y de decidir si aceptar o rechazar estos mensajes. También define el formato de mensaje de red que permite que las PDU de la capa de transporte sean transportadas por la *Bearer Layer*.

- *Lower Transport Layer*:

  La *Lower Transport Layer* toma las PDUs de la capa *Upper Transport*. Si estas cuentan con más de 31 bytes, esta capa realiza una segmentación para dividirlas en PDUs más pequeñas, de modo que este proceso permite enviar mensajes largos a través del portador de publicidad. En el momento en el que la capa *Lower Transport* de otro dispositivo reciba la PDU segmentada, la volverá a montar en una sola PDU y se reenviará a las capas superiores de la pila.

- *Upper Transport Layer*:

  Esta capa es la responsable del cifrado, descifrado y autenticación de los datos de la aplicación que pasan hacia y desde la capa de acceso.

[15] GAP (*Generic Access Profile*) es una capa de la pila del protocolo Bluetooth Low Energy que está a cargo de controlar conexiones y anuncios en Bluetooth. GAP es el mecanismo que hace visible un dispositivo compatible con BLE al mundo exterior, y determina el modo en el que dos dispositivos pueden (o no pueden) interactuar entre ellos.

- *Access Layer*:

  La *Access Layer* es responsable de definir mecanismos sobre cómo las aplicaciones pueden hacer uso de la capa *Upper Transport*. Esto incluye la definición del formato de los datos de la aplicación, la definición y control del proceso de cifrado y descifrado y, por último, la verificación de que los datos recibidos de la *Upper Transport Layer* sean para la red y la aplicación correctas antes de reenviar los datos a la pila.

- *Foundation Model Layer*:

  Es la capa que define los estados, modelos y mensajes necesarios para administrar y configurar una red en malla.

- *Model Layer*:

  En las redes Bluetooth Mesh, un modelo representa un caso de aplicación específico estandarizado, por lo que esta capa es la responsable de la definición de los modelos, y como tal, de la implementación de sus comportamientos.

3.2. Aprovisionamiento y configuración

Como se ha mencionado anteriormente, los nodos son dispositivos Bluetooth especialmente preparados. Esto se debe a que, si un dispositivo quiere unirse a una determinada red en malla, primero debe pasar por un proceso de seguridad preparatorio llamado *aprovisionamiento*.

Este aprovisionamiento se realiza manualmente por un administrador de red, por ejemplo, un smartphone, el cual equipa al dispositivo no aprovisionado con una serie de claves de seguridad. Una vez aprovisionado, al dispositivo ya se le conoce como nodo y puede comunicarse con otros nodos de la red.

Aunque en la sección 3.6.5. de este capítulo se profundizará en el protocolo de aprovisionamiento, se puede resumir este proceso en cinco pasos para ir entrando en materia:

i. *Paso 1:* el dispositivo (*slave*) no aprovisionado transmite paquetes de publicidad.

ii. *Paso 2:* una vez que el aprovisionador (*master*) recibe los paquetes publicitarios del dispositivo (*slave*), envía una PDU de invitación de aprovisionamiento y el *slave* responde con su información específica, como, por ejemplo, el fabricante y su dirección MAC.

iii. *Paso 3:* se produce el intercambio de las claves públicas entre el dispositivo no aprovisionado y el aprovisionador.

iv. *Paso 4:* el dispositivo no aprovisionado genera números aleatorios que deben ser ingresados en el aprovisionador por el usuario.

v. *Paso 5:* se crean las claves de sesión obtenidas de las claves públicas (del paso 3) y las claves privadas del dispositivo no aprovisionado y del aprovisionador. Esta clave de sesión asegura el envío de los recursos necesarios para completar el proceso de aprovisionamiento. Por tanto, los recursos que configuran los nodos y permiten a estos unirse a una red de malla Bluetooth son:

- *Address*: se necesitan direcciones de red para identificar las fuentes y los destinos de los mensajes.

- *NetKey*: clave de red generada aleatoriamente para proteger y autenticar mensajes en la capa de red. Se comparte entre todos los nodos de la red.

- *AppKey:* las claves de aplicación también se generan de manera aleatoria y se comparten entre los nodos de una red que participan en una aplicación determinada.

- *IV index*: es un vector de inicialización de 32 bits que comparten todos los nodos de una red. Su propósito es generar aleatoriedad en el cálculo de los valores de *Nonce*[16] del mensaje.

3.3. Tipos de nodos

Los nodos, además de tener capacidad para enviar y recibir mensajes, pueden poseer diferentes características que permiten que cada uno de estos desempeñe un papel esencial en la red. A continuación, se listan los diferentes roles que pueden desempeñar:

- *Relay*:

  Se trata de un nodo que recibe y retransmite mensajes dentro de la malla Bluetooth a través del portador de publicidad (*advertising bearer*). Desempeña el papel principal de extender el alcance de la comunicación de la red en malla Bluetooth puesto que, en el proceso de retransmisión, hace posible la comunicación *multi-hop*.

  Las PDUs de la red en malla incluyen un campo llamado TTL (*Time To Live*), el cual extiende o limita la vida útil de la red. Este campo toma un valor entero y marca el número de saltos que, como máximo, hará un mensaje por la red.

- *Proxy*:

  Este nodo tiene la capacidad de intercambiar mensajes entre *advertising bearer y GATT bearer*, puesto que exponen una interfaz GATT que los dispositivos Bluetooth LE pueden utilizar para interactuar con la red en malla. Por tanto, gracias a los nodos *proxy*, es posible que los dispositivos BLE que no tienen la pila del

[16] *Nonce* es un número que solo puede usarse una vez, es decir, en una red Bluetooth Mesh, cada vez que se cifra un mensaje, a este se le asigna un nuevo valor de *Nonce*. En la seguridad de las comunicaciones, se puede usar como forma de protección contra ataques de *replay*.

protocolo Bluetooth Mesh, se comuniquen con dispositivos que forman parte de una red en malla Bluetooth.

- *Low Power*:

  Este nodo se caracteriza por admitir y habilitar la función de baja potencia, por lo que ayuda a los nodos que funcionan con baterías, minimizar su consumo tanto como sea posible. El nodo de baja potencia puede funcionar en una malla Bluetooth con la ayuda de un nodo *Friend*, ya que este tipo de nodo puede funcionar como un dispositivo apagado y solicitar los mensajes almacenados por su nodo amigo asociado durante su periodo de apagado. El proceso de soportar un nodo *Low Power* por un nodo *Friend*, se conoce como *friendship*.

- *Friend*:

  Friendship es un nuevo concepto introducido por el estándar Bluetooth Mesh, y como bien se mencionaba antes, se diseñó para incluir dispositivos con limitación de energía. Este tipo de nodo se encarga de almacenar y reenviar mensajes destinados a su nodo *Low Power* asociado.

En la siguiente figura (Figura 3), se puede observar un ejemplo de topología en malla con los diferentes tipos de nodos en una red Bluetooth Mesh totalmente aprovisionada, es decir, todos los dispositivos han sido aceptados en la red y se les han asignado tanto las direcciones correspondientes como las claves de red.

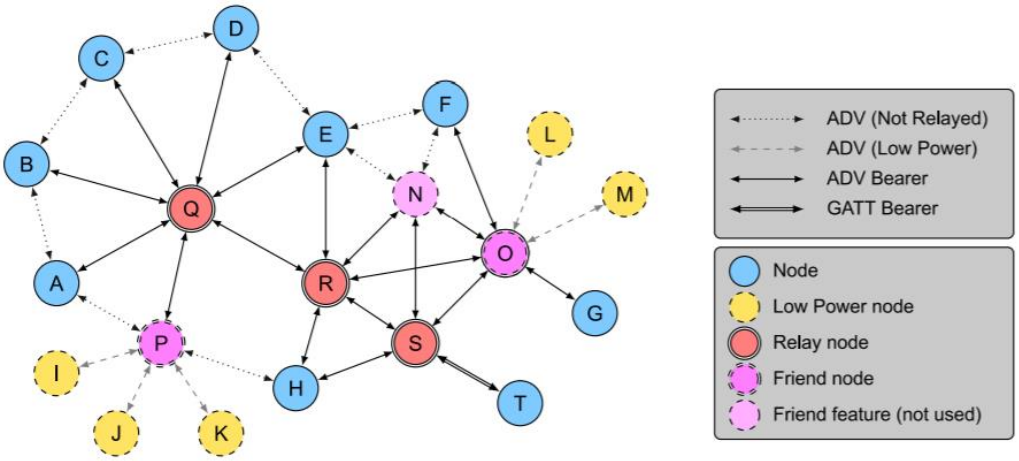

Figura 3. Ejemplo de topología de red en malla Bluetooth [21]

3.4. Composición de los nodos

Los nodos están compuestos de *elementos*, *modelos* y *estados*. Antes de profundizar en cada uno de estos componentes, se puede observar en la siguiente figura cómo es la jerarquía que conforma a un determinado nodo.

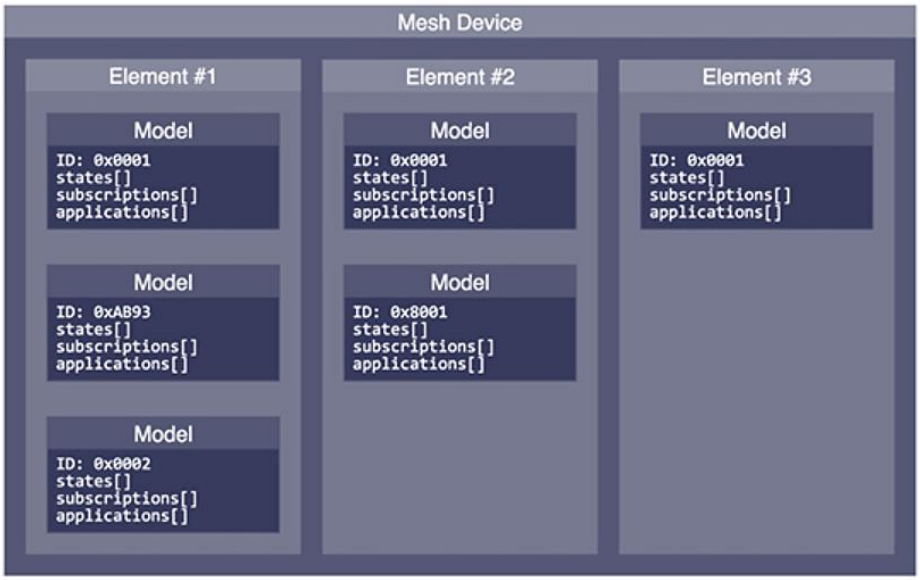

Figura 4. Composición de un nodo [20]

3.4.1. Elementos

Dentro de Bluetooth Mesh surge el concepto de elemento, que corresponde a una parte individual dentro de un nodo. Cada elemento está direccionado y es el primero de estos elementos (*primary element*) el que contiene los datos de configuración que son aplicables a todo el nodo, lo cual significa que todos los nodos deben tener al menos un elemento.

3.4.2. Modelos

Como se ha visto, los nodos contienen uno o más elementos, y cada uno de ellos contiene uno o más modelos. Los modelos pueden ser considerados como los componentes software que hacen que un nodo sea capaz de realizar ciertos comportamientos o presenten un servicio específico. Se podría decir que en Bluetooth Mesh, los modelos sustituyen la función de los perfiles[17] en Bluetooth Low Energy.

Los modelos pueden ser definidos y adoptados por Bluetooth SIG, y se conocen como *SIG adopted models*, pero también pueden ser definidos por los proveedores, que reciben el nombre de *vendor models*. Los modelos se identifican mediante identificadores únicos, que pueden ser de 16 bits, para los *SIG adopted models*, o de 32 bits, para los *vendor models*.

Por otro lado, es importante destacar que la comunicación en Bluetooth Mesh se basa en la arquitectura *cliente-servidor*, por lo que hay que mencionar que se utilizan dos categorías de modelos, definidas como:

[17] Los *perfiles* BLE simplemente exponen colecciones de *servicios* predefinidas por Bluetooth SIG, los cuales a su vez son colecciones de *características* que encapsulan el comportamiento de un dispositivo.

- *Server model:*

  Consta de uno o varios estados que cubren uno o más elementos. Además, define un conjunto de mensajes y el comportamiento del elemento al enviar o recibir dichos mensajes.

- *Client model:*

  No tiene ningún estado definitivo, sin embargo, determina los mensajes que el cliente puede enviar para obtener, cambiar o usar los estados del servidor correspondiente.

Además, existe un modelo obligatorio en todo elemento primario del nodo. Se trata del modelo *Configuration Server*, el cual contiene valores de estado que constituyen la configuración del nodo. Este modelo no puede asociarse a un elemento que no sea el primario, lo que hace que dicho elemento principal sea especial y diferente de otros elementos que podría tener un nodo.

La especificación de los modelos de malla de Bluetooth SIG [22] define de manera rigurosa y extendida un conjunto de hasta 52 modelos de malla estándar (Figura 5).

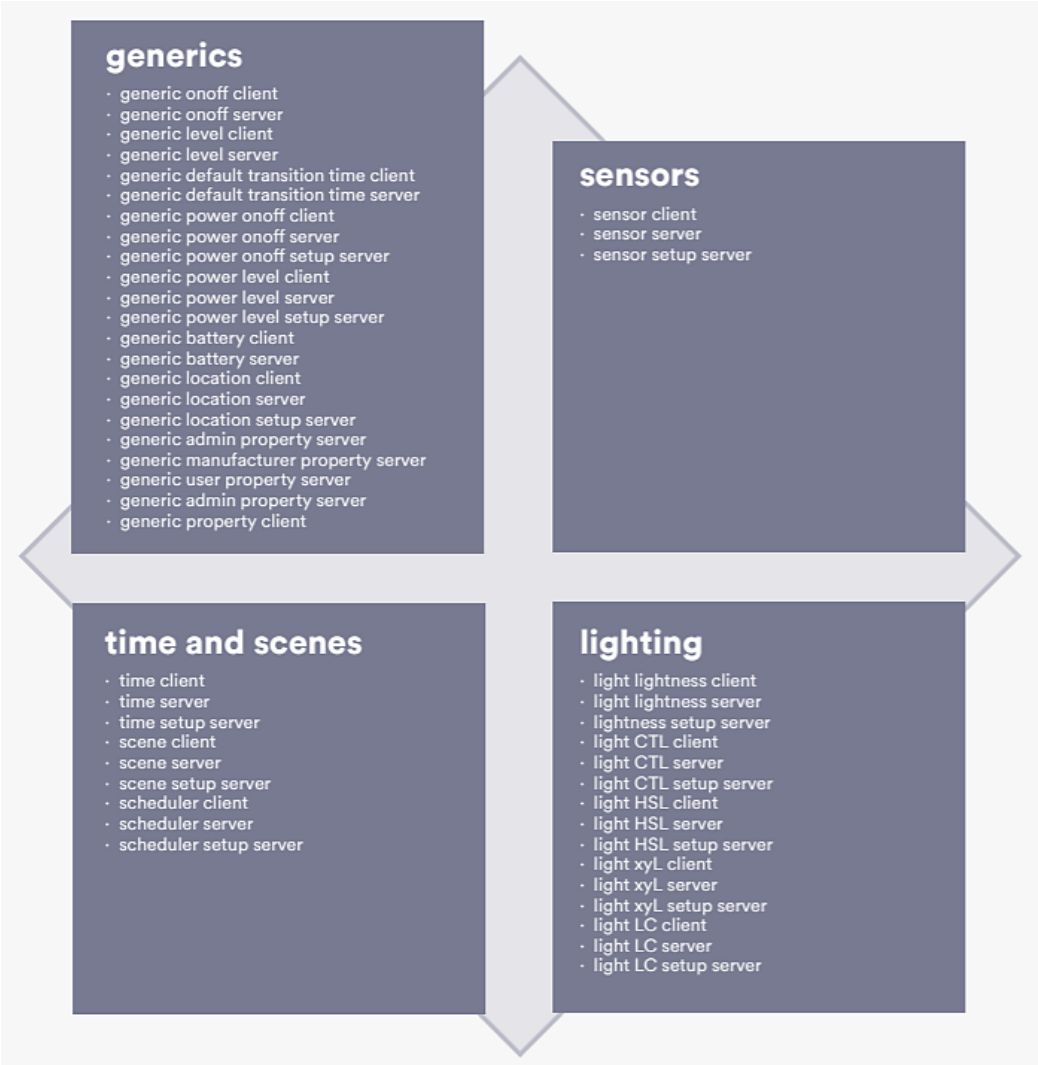

Figura 5. Modelo Estándar Bluetooth Mesh [23]

Como se puede ver en la Figura 5, hay cuatro grupos de modelos. En primer lugar, estarían los modelos genéricos, después, estarían los modelos para sensores y para la iluminación, y, por último, los modelos relacionados con el tiempo y una función de automatización de malla llamada escena. De manera general, los modelos son opcionales, ya que los desarrolladores pueden implementar aquellos modelos que equipen a sus productos con las capacidades de malla que necesiten.

La colección de modelos *generics* de Bluetooth Mesh está diseñada para ser utilizada por cualquier tipo de dispositivo, pues dichos modelos ofrecen un conjunto de capacidades comunes y generalmente aplicables. Si se revisan las listas de la Figura 5, también se tiene que para cada modelo cliente está asociado a un modelo servidor y viceversa.

El modelo *Generic On/Off Server* es uno de los modelos ideales para empezar a trabajar con esta tecnología de red, pues permite que el nodo que lo implemente pueda ser encendido o apagado por otro nodo que contenga el modelo *Generic On/Off Client*. Por su parte, los modelos *Generic Level Client* y *Server* permiten ejercer al nodo que los implemente control sobre el nivel de otros dispositivos. Los modelos *Generic Power On/Off Client* y *Server* permiten configurar el estado inicial en el que se encuentra un dispositivo inmediatamente después de encenderlo. Por ejemplo, en algunos casos es preferible que el estado inicial de un dispositivo cuando se enciende sea apagado, indicado por un valor de *0x00* en el estado. Alternativamente, para otro producto, puede tener más sentido que el estado inicial esté encendido, con el estado establecido en *0x01*. Por otro lado, el modelo *Generic Battery Server* representa a un elemento que funciona con batería, mientras que el cliente de este mismo modelo se puede utilizar para monitorear el estado de los elementos alimentados por batería. Por otro lado, a veces resulta útil saber dónde se encuentra un determinado dispositivo de la red. Los modelos *Generic Location Client* y *Server* permiten hacer esto. Por último, a pesar de que quedan modelos genéricos por mencionar, es interesante tratar los modelos *Generic Property Client* y *Server*, que permiten que se asocien listas de números arbitrarios de propiedades con un dispositivo. Las propiedades se agrupan en diferentes modelos para que los diferentes grupos de usuarios, como, por ejemplo, el fabricante, el usuario o el administrador de la red, solo tengan acceso a las propiedades permitidas. También es posible que un modelo *Generic Property Server* encuentre un cliente que sea capaz de consumir y utilizar un tipo de propiedad en particular. En conjunto, los modelos *Generic Property* proporcionan un mecanismo de almacenamiento de datos y comunicación que puede adaptarse a una amplia gama de valores y tipos de datos sin que sea necesario cambiar los modelos.

Por su parte, la iluminación puede ser sorprendentemente sofisticada y, por lo tanto, necesita modelos de la malla Bluetooth especializados para cumplir con sus requisitos. Los modelos *lighting* de Bluetooth Mesh permiten controlar el estado de encendido o apagado de las luces, su luminosidad, su color, etc.

Las luces a menudo se controlan manualmente, pero también pueden ser controladas por sensores o por temporizadores. El modelo *Generic On/Off*, anteriormente mencionado, podría usarse para controlar algunos de los atributos básicos de la luz; sin embargo, las luces disponen de atributos más complejos que se podrían controlar. Aquí es donde entra

en juego el término *Smart lighting*, pues los modelos de iluminación de Bluetooth Mesh incluyen un conjunto de modelos particularmente especiales, como son los modelos *Light LC*, que proporcionan un control sofisticado y automatizado de las luces. Estos últimos modelos (*lighting controller*) conforman un controlador de iluminación. Se trata de un componente software que permite configurar un control de iluminación refinado, controlado por sensores y por el usuario. A medida que cambia el estado del modelo *Light LC*, el estado de luminosidad de la luz progresa a través de una serie de niveles, los cuales están programados con unas transiciones en las que los cambios nos son abruptos, pues se quiere que estos sean naturales para los usuarios de, por ejemplo, el edificio donde se ha instalado esta solución BLE Mesh.

Por otro lado, las escenas (*scenes*) de la malla Bluetooth definen colecciones completas de configuraciones para un entorno con el objetivo de optimizarlo para un propósito particular. El cambio a un tipo de escena concreto puede activarse mediante sensores o mediante un horario, ya que, para respaldar las operaciones programadas, la malla Bluetooth hace posible que se propague a una hora precisa a todos los nodos de la red las instrucciones para que cambien a los estados que pertenecen a esa escena específica. Por su parte, los sensores detectan e informan de eventos, como, por ejemplo, el estado cambiante de la ocupación de las habitaciones. Los datos del sensor se pueden usar para influir o controlar el funcionamiento de un tipo particular de dispositivo, o para cambiar el estado de múltiples dispositivos de diferentes tipos, todo de una vez. Gracias a los modelos *time* y *scenes* y los modelos *sensors* se puede conseguir este tipo de aplicaciones.

Finalmente, hay que mencionar que los modelos se comunican entre sí a través de un sistema de *publicación-suscripción*. El envío de mensajes desde un nodo a otro nodo o a un conjunto de nodos se denomina publicación, mientras que la suscripción consiste en la configuración de un nodo para recibir estos mensajes enviados a través de direcciones específicas para su procesamiento.

3.4.3. Estados

El estado no es más que un valor, operado por mensajes, que se utiliza para describir la condición de un elemento. Por ejemplo, si hay una lámpara, su estado será encendido o apagado.

3.5. Mensajes y direccionamiento

La red de malla se comunica a través de dos tipos de mensajes diferentes. Primero están los *mensajes de control*, que son mensajes relacionados con la malla, es decir, indican que un nodo particular está presente, el número de saltos, etc. Y después están los *mensajes de acceso*, que están relacionados con la funcionalidad de cada aplicación. Estos últimos se dividen en tres tipos:

- *Mensajes Set:*

  Modifican el estado de los nodos. Hay dos variantes de este tipo de mensajes. Por un lado, están los mensajes *acknowledged* o ACK, respondidos por un mensaje *Status* que contiene el valor del nuevo estado, y por otro lado están los mensajes *unacknowledged* o NACK, que no son respondidos.

- *Mensajes Get:*

  Solicitan y recuperan el estado actual de un nodo, el cual responde con un mensaje *Status*.

- *Mensajes Status:*

  A través de estos mensajes, los nodos anuncian el valor actual de un estado determinado.

Para el direccionamiento de estos mensajes, existen tres tipos de direcciones:

- *Direcciones unicast:*

  Se trata de una dirección de 16 bits única que identifica de forma exclusiva a un único elemento en un nodo durante el proceso de aprovisionamiento. El número máximo de direcciones de unidifusión, y, por tanto, el número máximo de elementos que una red en malla puede tener es 32767.

- *Direcciones de grupo:*

  Es una dirección multidifusión que representa uno o más elementos o a uno o varios nodos. Cuando se envía un mensaje desde un nodo, es decir, cuando el nodo publicita un mensaje a una dirección de grupo, todos los elementos suscritos a ese grupo recibirán el mensaje. En total, hay 16384 direcciones de grupo disponibles.

- *Direcciones virtuales:*

  Estas direcciones también identifican colecciones de dispositivos como las direcciones de grupo. Se diferencian de estas últimas en que se asigna una dirección de 16 bits a un UUID de 128 bits. Las direcciones virtuales son menos eficientes en términos de procesamiento en tiempo de ejecución.

Por otro lado, recordando que la comunicación se produce a través del sistema *publicación-suscripción*, es importante mencionar que los mensajes tienen una dirección origen y una dirección destino. Las direcciones de origen son siempre direcciones *unicast*, mientras que las direcciones destino pueden ser de cualquiera de los tres tipos de direcciones mencionados anteriormente.

Finalmente, como se vio anteriormente en el apartado 3.1. de este capítulo, la capa de transporte del protocolo Bluetooth Mesh además de gestionar la segmentación de mensajes cuando estos alcanzan el tamaño máximo, también se encarga de la encriptación de estos.

A través de la Figura 6, se puede obtener una idea gráfica del formato de estos paquetes BLE Mesh sin necesidad de profundizar mucho más en ello.

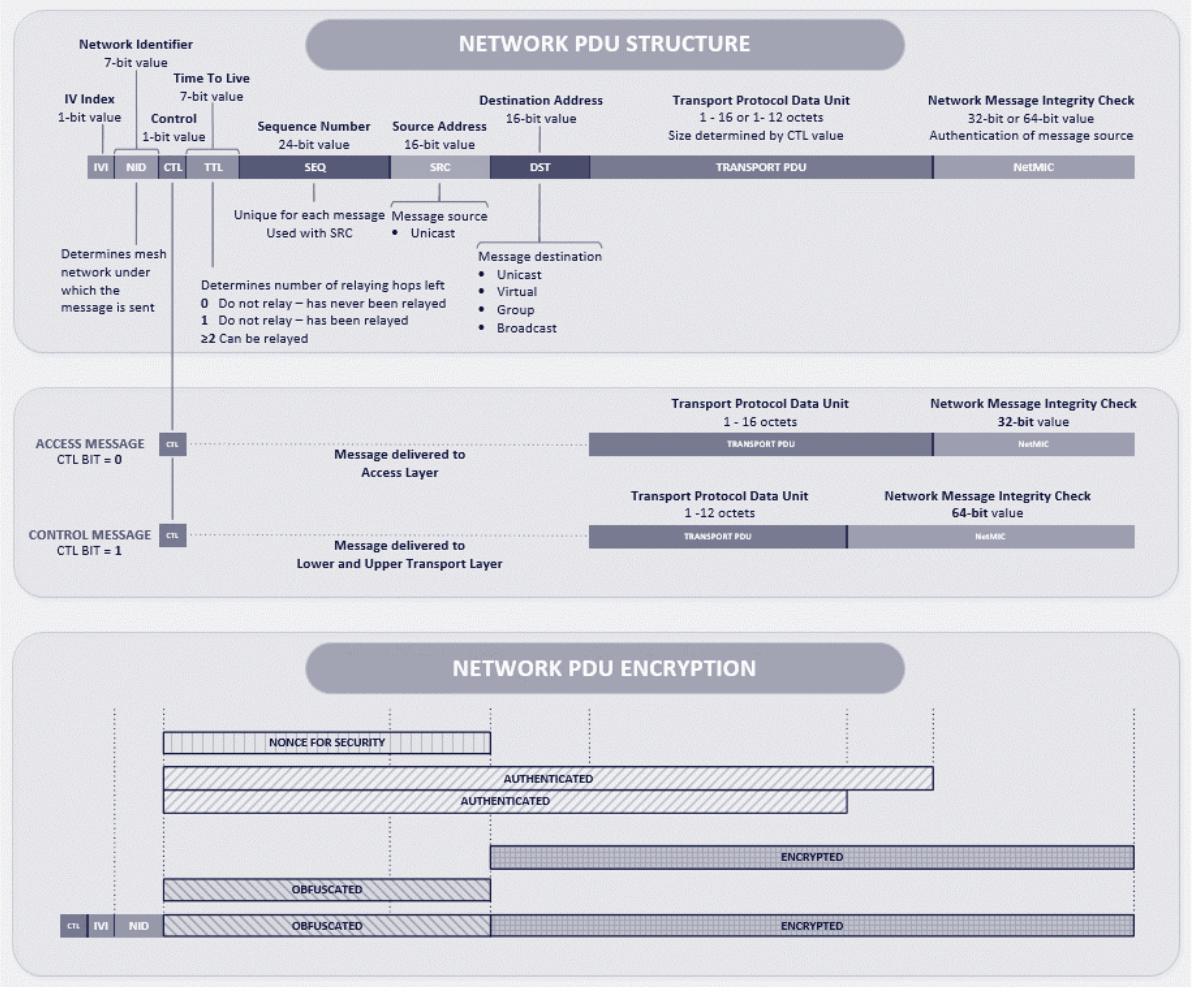

Figura 6. Formato del paquete en la malla Bluetooth (PDU) [19]

3.6. Seguridad

La seguridad en las nuevas tecnologías y en las ya existentes, es una preocupación recurrente, especialmente si tratamos con el Internet de las Cosas (IoT). A diferencia de BLE, donde la seguridad es opcional para proteger a un solo dispositivo, la malla Bluetooth [24] la impone de forma obligatoria, ya que organizar una gran red con muchos dispositivos plantea un enorme riesgo en la transferencia segura de datos.

3.6.1. Fundamentos

En la siguiente tabla se recogen las principales características de seguridad de Bluetooth Mesh.

Cifrado y autenticación	Todos los mensajes están encriptados y autenticados.
Separación de intereses	La seguridad del dispositivo, la seguridad de la red y la seguridad de las aplicaciones se abordan de forma independiente (*Separation of Concerns*).
Aislamiento de área	Una red de malla Bluetooth se puede dividir en subredes, cada una criptográficamente distinta y segura de las demás (*NetKey*).
Actualización de clave	Las claves de seguridad se pueden cambiar durante la vida útil de la red Bluetooth Mesh mediante un procedimiento de actualización de clave.
Ofuscación de mensajes	La ofuscación de mensajes dificulta el seguimiento de los mensajes enviados dentro de la red y, como tal, proporciona un mecanismo de privacidad para dificultar el seguimiento de los nodos.
Protección contra ataques de *replay*	La propia seguridad de Bluetooth Mesh protege la red contra este tipo de ataques.
Protección contra ataques *Trashcan*	Al eliminar los nodos de la red de forma segura, se evitan los ataques *Trashcan*
Aprovisionamiento seguro de dispositivos	El proceso mediante el cual los dispositivos se agregan a la red de malla Bluetooth para convertirse en nodos.

Tabla 2. Fundamentos de la seguridad en Bluetooth Mesh [24]

3.6.2. Criptografía

La mayor parte de funciones de seguridad de la red Bluetooth en malla, se basan en algoritmos criptográficos y procedimiento estándar de la industria. En Bluetooth Mesh hay dos algoritmos de seguridad fundamentales de cifrado y autenticación, de los cuales se va a hablar brevemente a continuación.

- *AES-CMAC*:

  El algoritmo *CMAC* (*Cipher-based Message Authentication Code*) tiene excelentes capacidades de detección de errores, ya que está diseñado tanto para detectar modificaciones intencionadas y no autorizadas como modificaciones accidentales. Puede generar un valor de autenticación de mensaje de una longitud fija de 128 bits para cualquier entrada de longitud variable. La fórmula para generar un código de autenticación de mensaje (*MAC*) es:

  $$MAC = AES\ CMAC_k(m)$$

  Donde k es la clave de 128 bits y m son los datos de longitud variable que se autenticarán.

- *AES-CCM*:

  En Bluetooth Mesh, el algoritmo *CCM* (*Cipher block chaining Message Authentication Code*) se utiliza como la función básica de cifrado y autenticación, ya que proporciona confidencialidad durante la transferencia de datos. La fórmula para su uso es:

  $$MIC = AES\ CCM_k(n, m, a)$$

  Donde *k* es la clave de 128 bits, *n* es un *Nonce*, *m* son los datos de longitud variable que se cifrarán y autenticarán y *a* son los datos de longitud variable que se autenticarán, pero no se cifrarán (*additional data*).

  Después de tomar cuatro entradas, este algoritmo da como resultado dos salidas. Por un lado, el texto cifrado y el valor de verificación de integridad del mensaje (*MIC*).

3.6.3. Separación de conceptos (*separation of concerns*)

Dentro del esquema de seguridad del proceso de aprovisionamiento, se introduce el concepto de separación de conceptos, en el que hay tres tipos de claves que proporcionan seguridad en diferentes aspectos de la red de Bluetooth Mesh.

- Clave del dispositivo *(DevKey):*

  En el proceso de aprovisionamiento, el aprovisionador asigna las *DevKeys* a cada nodo de la malla, lo cual permite la identificación única de nodos de la malla. El aprovisionador usa la clave del dispositivo solo durante el proceso de configuración del nodo.

- Clave de red (*NetKey*):

  Un nodo puede poseer una o más claves de red. Esto permite la creación de múltiples subredes dentro de una única red Bluetooth Mesh. Esta partición evita la retransmisión de mensajes en todos los niveles y limita los mensajes retransmitidos al nivel deseado. Por tanto, la posesión de una determinada *NetKey* es lo que define la permanencia de un nodo a una determinada red o subred de la malla.

- Clave de aplicación (*AppKey*):

  El aprovisionador es el encargado de generar y distribuir las *AppKeys*. Estas se comparten entre un subconjunto de dispositivos dentro de una red de malla. En general, estos dispositivos tienen una funcionalidad similar.

3.6.4. Protección contra ataques

La seguridad en la malla Bluetooth protege a la red contra diversas amenazas en múltiples capas. A continuación, se estudiarán en detalle tres ataques resueltos por la propia seguridad de esta tecnología.

- Ataque *Man-In-The-Middle*:

  Este famoso ataque se daba en Bluetooth LE cuando un usuario quería realizar el proceso de emparejamiento entre dos dispositivos, pero en lugar de conectarlos directamente, estos sin saberlo se estaban conectando a un tercer dispositivo ilícito, el cual de manera engañosa representaba el papel de un dispositivo con el que ellos querían emparejarse. Por tanto, este dispositivo ilícito tendría la habilidad de desviar y controlar las comunicaciones entre los dos dispositivos principales.

  En Bluetooth Mesh se resuelve este ataque puesto que durante el aprovisionamiento se usa el protocolo de intercambio de claves ECDH[18] (*Elliptic Curve Diffie-Hellman*), lo cual hace que se agreguen dispositivos confiables a la red.

- Ataque de *replay*:

  Este tipo de ataque se da cuando un hacker intercepta y captura uno o más mensajes entre la comunicación de dos dispositivos y más tarde los retransmite con el objetivo de engañar al destinatario para que realice algo que el dispositivo del atacante no está autorizado a hacer.

  En Bluetooth Mesh se aborda este ataque mediante el uso del número de secuencia (*SEQ*) y el índice de vector de inicialización (*IV Index*). Los elementos de los nodos incrementan el valor de *SEQ* cada vez que publican un mensaje, por tanto, un nodo que recibe un mensaje de un elemento que contiene un valor de *SEQ* menor o igual que el último mensaje válido lo descartará, ya que probablemente esté relacionado con un ataque de *replay*. Del mismo modo, los valores del *IV Index* dentro de los mensajes de un elemento, siempre deben ser igual o mayor que el último mensaje válido de ese elemento.

- Ataque *Trashcan*:

  En el momento en el que un nodo se vuelva defectuoso y deba eliminarse o simplemente el propietario quiere quitarlo de su red para, por ejemplo, venderlo, es importante que el dispositivo en cuestión y las claves que contiene no puedan usarse para montar un ataque en la red de la que se extrajo el nodo.

  El ataque *trashcan* se evita eliminando los nodos de la red de manera segura. Para ello, se agrega el nodo a una "lista negra" y después se inicia el procedimiento de actualización de claves.

  Este procedimiento emite a todos los nodos de la red, excepto a los que son miembro de la "lista negra", nuevas *NetKeys* y *AppKeys*, es decir, se reemplaza todo

[18] Acuerdo de claves anónimo que permite que dos partes, las cuales tienen tanto clave pública como privada, puedan establecer un secreto compartido sobre un canal inseguro. Este secreto compartido o clave se puede utilizar para cifrar comunicaciones subsiguientes usando un sistema de cifrado de clave simétrica.

el conjunto de claves de seguridad. Por tanto, el nodo que se eliminó de la red y contiene antiguas *NetKeys* y *AppKeys*, ya no representa una amenaza.

3.6.5. Protocolo de aprovisionamiento

El proceso de aprovisionamiento consiste en agregar un nuevo dispositivo (no aprovisionado) a una red Bluetooth Mesh. Se podría decir que este proceso es la base de la seguridad en Bluetooth Mesh, puesto que permite que los dispositivos se comuniquen de manera confiable y segura.

Por otro lado, el protocolo de aprovisionamiento define las PDUs que usan para comunicarse un aprovisionador y un nuevo dispositivo no aprovisionado durante el proceso de aprovisionamiento. En dicho proceso, el aprovisionador, proporciona al dispositivo (*slave*) datos que le permiten convertirse en un nodo de la malla Bluetooth. Como se comentó en el apartado 3.2 de este capítulo, ese procedimiento consta de cinco fases, las cuales se van a describir a continuación de manera mucho más detallada y teniendo en cuenta la seguridad que engloba [25][26].

i. *Beaconing* (balizamiento):

Esta primera fase funciona igual que en Bluetooth Low Energy, es decir, se usa el mecanismo de *advertising*.

Si un dispositivo (no aprovisionado) admite el portador *PB-ADV* (*advertising bearer*), se anuncia como una baliza y permite que un dispositivo aprovisionador lo descubra. Sin embargo, cuando un dispositivo sin aprovisionamiento utiliza el portador *PB-GATT* (*GATT bearer*), un servicio GATT llamado *Mesh Provisioning Service* respalda el procedimiento general de aprovisionamiento. Cuando el dispositivo transmite paquetes de *advertising*, que incluyen el UUID del servicio anteriormente nombrado, el aprovisionador lo descubre a través del escaneo estándar BLE.

ii. *Invitation* (invitación):

En este segundo paso, el aprovisionador envía una PDU de invitación (*provisioning invite*) y el dispositivo esclavo responde con una PDU de capacidades (*provisioning capabilities*).

La PDU de *provisioning invite*, incluye un campo que indica cuánto tiempo el elemento primario del dispositivo no aprovisionado debería atraer la atención del usuario, por ejemplo, con alguna indicación visual, mientras que la PDU de *provisioning capabilities* incluye información como el número de elementos que admite el dispositivo, la capacidad de este para generar un valor para el usuario o para permitir que el usuario ingrese un valor, etc.

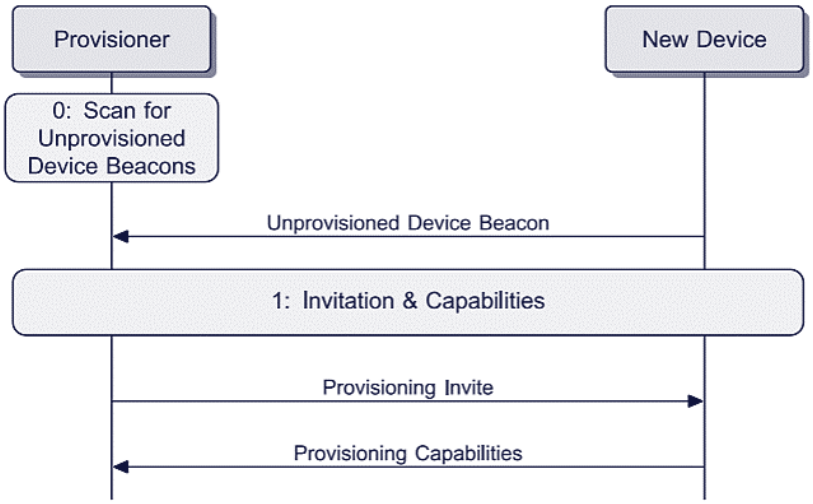

Figura 7. Provisioning invitation [21]

Por lo tanto, en esta fase de invitación, el objetivo es proporcionar al aprovisionador información sobre las capacidades del dispositivo no aprovisionado.

iii. *Exchange Public Keys* (intercambio de claves públicas):

Antes de ver cómo se produce el intercambio de claves públicas en esta tecnología, es necesario saber que existen dos técnicas para cifrar información:

- Cifrado simétrico (cifrado de clave secreta):

  Utiliza la misma clave secreta para el cifrado y descifrado. Siempre que el remitente y el destinatario conozcan la clave secreta, pueden descifrar todos los mensajes cifrados con esta clave. El problema de esto es que es difícil intercambiar claves secretas de forma segura a través de un enlace, ya que pueden caer en manos equivocadas.

- Cifrado asimétrico (cifrado de clave pública):

  Utiliza un par de claves relacionadas: la clave pública y la clave privada. La clave pública se pone a disposición de cualquier persona que desee enviarle un mensaje. La clave privada se mantiene en secreto, para que solo el dispositivo correspondiente la conozca. Cualquier mensaje cifrado con la clave pública solo se puede descifrar utilizando la clave privada correspondiente. Esto significa que no hay que preocuparse por pasar claves públicas a través del enlace, ya que solo se utilizan para el cifrado y no para el descifrado. Sin embargo, a diferencia del anterior cifrado, el cifrado asimétrico es más lento y requiere mucha más potencia de procesamiento para cifrar y descifrar el contenido de los mensajes.

  Bluetooth Mesh utiliza una combinación de métodos asimétricos y simétricos para resolver el problema de la potencia del procesador requerida por parte de los dispositivos Bluetooth LE.

- Criptografía asimétrica:

  Se usa el algoritmo de *Elliptic Curve Diffie-Hellman* (ECDH), que permite un anónimo acuerdo entre dos partes, las cuales cuentan con par de claves público-privada, que pueden establecer un secreto compartido sobre un canal inseguro. Por tanto, el propósito de ECDH es crear un enlace seguro entre el aprovisionador y el dispositivo no aprovisionado, en el que los dos dispositivos pueden usar la clave simétrica para cifrar y descifrar mensajes posteriores.

- Criptografía simétrica:

  Cada mensaje transmitido en una red de malla Bluetooth se encripta usando la criptografía *AES-128*[19]. El algoritmo *AES-128* es un motor de cifrado/descifrado simétrico común utilizado a menudo en plataformas integradas.

Tras hablar sobre las técnicas que existen para cifrar información, es hora de tratar el tema principal de esta fase, que es el intercambio de claves públicas. Por un lado, existe el intercambio de claves públicas a través de un túnel OOB[20] y por otro, a través de un enlace Bluetooth.

- Enlace Bluetooth:

  En la fase de invitación, el dispositivo no aprovisionado informa de que no admite enviar su clave pública a través de un túnel OOB. Por tanto, ambas claves públicas se intercambian a través de un enlace Bluetooth.

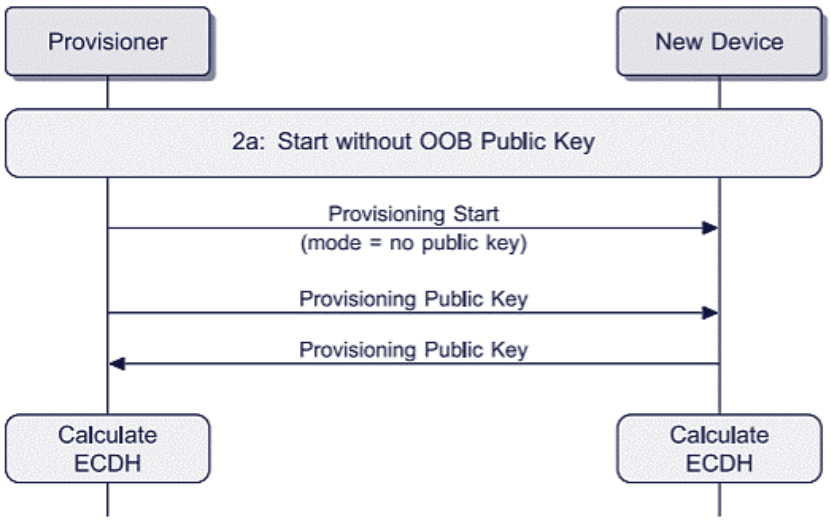

Figura 8. Intercambio de clave pública cuando se desconoce la clave pública del dispositivo no aprovisionado [21]

[19] *Advanced Encryption Standard (AES)*. https://www.nist.gov/publications/advanced-encryption-standard-aes.
[20] *Out-Of-Band* es un modelo que recurre a tecnologías alternativas a Bluetooth.

- Túnel OOB:

  En la fase de invitación, el dispositivo no aprovisionado informa que admite enviar su clave pública a través de un túnel OOB. Por tanto, se transmite una clave pública efímera desde el aprovisionador al dispositivo y se lee una clave pública desde el dispositivo no aprovisionado utilizando una tecnología OOB adecuada, por ejemplo, con un código QR.

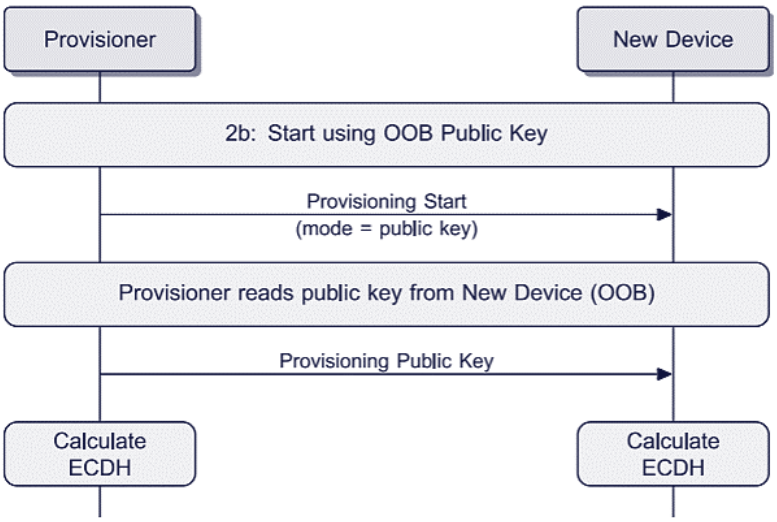

Figura 9. Intercambio de clave pública cuando el dispositivo no aprovisionado utiliza un método OOB [21]

iv. *Authentication* (autenticación):

En esta cuarta fase, el aprovisionador usa un método de autenticación para autenticar el dispositivo no aprovisionado. Los métodos pueden ser:

- *Output OOB*:

  Si se selecciona este método de autenticación, el dispositivo no aprovisionado elige un número aleatorio y genera ese número de manera compatible con sus capacidades. Por ejemplo, si el dispositivo no aprovisionado es una bombilla, podría parpadear un número determinado de veces. De esta forma, el usuario del aprovisionador ingresa el número observado para autenticar el dispositivo no aprovisionado.

  Después de ingresar el número aleatorio, el aprovisionador genera y verifica un valor de confirmación. La operación de *verificación del valor de confirmación* se describirá más adelante.

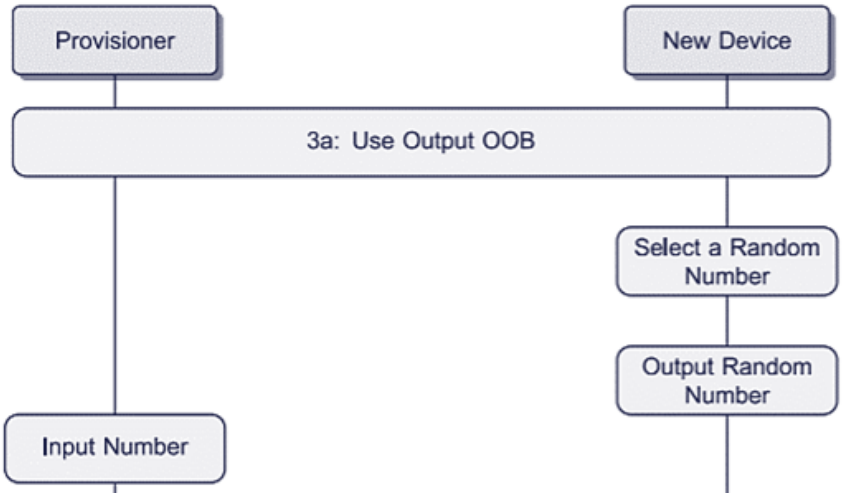

Figura 10. Autenticación usando el método Output OOB [21]

- *Input OOB*:

  Este método de autenticación es similar al método de *output OOB*, pero las funciones del dispositivo se invierten. El aprovisionador genera un número aleatorio, lo muestra, y luego solicita al usuario que ingrese el número aleatorio en el dispositivo no aprovisionado. Por ejemplo, un interruptor de luz puede permitir al usuario ingresar el número aleatorio presionando un botón un número determinado de veces dentro de un cierto período.

  El método *input OOB* requiere que se envíe una PDU adicional, ya que después de finalizar la acción de autenticación, el dispositivo sin aprovisionamiento envía una PDU *provisioning input complete* al aprovisionador para informarle que se ha ingresado el número aleatorio. El proceso continúa con la operación de *verificación del valor de confirmación*.

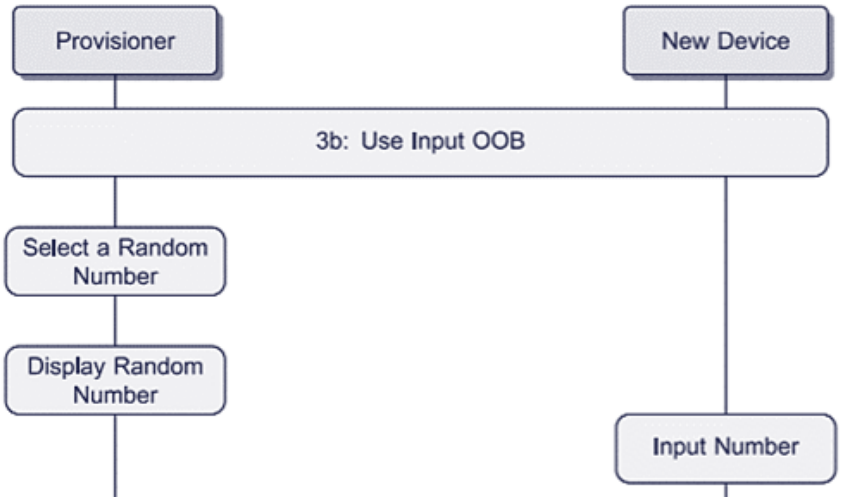

Figura 11. Autenticación usando el método Input OOB [21]

- *Static OOB o No OOB:*

  Se usa este método de autenticación cuando ni *output OOB* ni *input OOB* son posibles. En este caso, el aprovisionador y el dispositivo no aprovisionado generan un número aleatorio y luego proceden a la operación de verificación del valor de confirmación.

Figura 12. Autenticación usando el método Static OOB or No OOB [21]

Independientemente del método de autenticación utilizado, se lleva a cabo la generación y verificación del valor de confirmación. En primer lugar, el aprovisionador y el dispositivo sin aprovisionar deben calcular cada uno por separado un valor de confirmación. Los dos valores se conocen como *Confirmation Provisioner* y *Confirmation Device*. En la especificación de la malla Bluetooth [21], se ilustra el procedimiento del cálculo, el cual requiere de ocho parámetros. Después, para la generación del valor de confirmación se usa el algoritmo *AES-CMAC*.

Cuando los valores de confirmación están listos, los dos dispositivos se los intercambian y cada uno verifica la integridad del valor recibido.

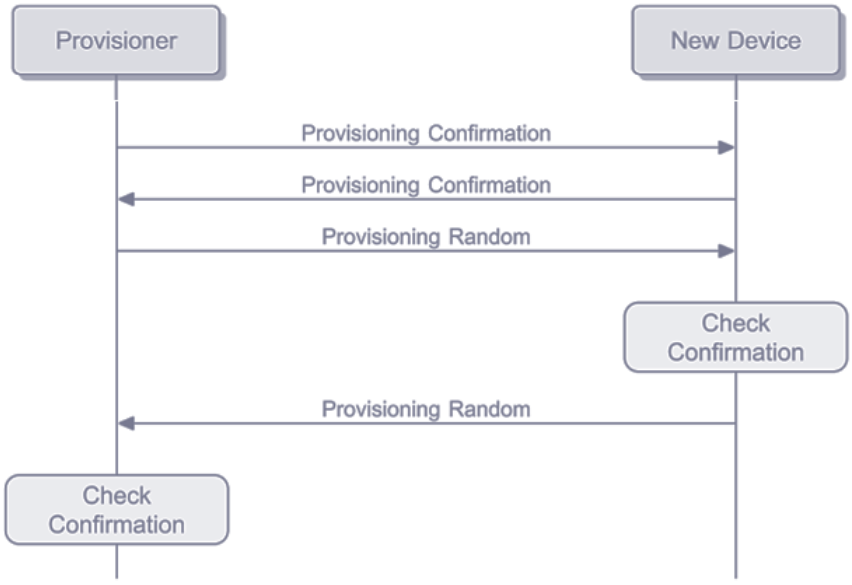

Figura 13. Verificación del valor de confirmación [21]

El proceso de confirmación comienza con el aprovisionador, que envía su número aleatorio, *Random Provisioner,* al dispositivo no aprovisionado. El dispositivo no aprovisionado lo usa para recalcular el *valor de confirmación* y verificarlo comparándolo con el *valor de confirmación* que recibió anteriormente. Si el *valor de confirmación* calculado por el dispositivo no coincide con el *Confirmation Provisioner* recibido, se cancela el proceso de aprovisionamiento. Sin embargo, si

coincide, el dispositivo no aprovisionado envía su valor de *Random Device* al aprovisionador. Después, el aprovisionador realiza el mismo proceso para recalcular el *valor de confirmación* y verificar comparando el valor calculado con el valor recibido previamente.

v. *Distribution of Provisoning Data* (distribución de datos de aprovisionamiento):

Una vez se ha completado el paso de autenticación, se procede a distribuir los datos de aprovisionamiento. El aprovisionador es responsable de generar los datos de aprovisionamiento, que consisten en la clave del dispositivo (*DevKey*), la clave de red (*NetKey*), *IV Index* y una dirección *unicast*.

Para distribuir los datos de aprovisionamiento de forma segura, el aprovisionador usa *AES-CCM* para cifrar los datos de aprovisionamiento con una *Session Key* y una *Session Nonce* compartida, que calculan tanto el aprovisionador como el dispositivo no aprovisionado. Cuando los valores *Session Key* y *Session Nonce* están listos, el aprovisionador cifra y envía la PDU de *provisioning data*, que contiene los datos de aprovisionamiento derivados al dispositivo no aprovisionado. Aquí, se utilizan los mismos valores *Session Key* y *Session Nonce* correspondientes para descifrar los datos recibidos.

Llegados a este punto, el proceso de aprovisionamiento ha finalizado. Los dos dispositivos conocen la nueva *DevKey* y la *NetKey* global, lo que hace que el nuevo dispositivo aprovisionado pase a nombrarse *nodo*, puesto que ya es miembro de la red Bluetooth Mesh.

Capítulo 4

Herramientas de desarrollo

Después de ver en el capítulo anterior los conceptos de la malla Bluetooth, es el momento de comenzar con el desarrollo de aplicaciones Bluetooth Mesh con kits de desarrollo reales, los cuales pueden proporcionar tanto la parte *hardware* como la *software*.

Como cualquier otro desarrollador, los desarrolladores de BLE Mesh se enfrentan a la pregunta de cómo saber qué kit de desarrollo se debe elegir para satisfacer las necesidades del proyecto a la hora de trabajar con esta tecnología. Es obvio que seleccionar la opción correcta puede llegar a ser un proceso complejo si no se tiene una idea clara del sistema que se quiere diseñar. Normalmente, no hay una diferencia significativa en la parte de desarrollo software, sin embargo, elegir la plataforma hardware puede ser algo más difícil, ya que hay cientos de chips Bluetooth Low Energy en el mercado.

Bluetooth SIG, con el propósito de ayudar a los desarrolladores, creó un documento [27] en el cual analiza de manera extensa los tres aspectos que considera más importantes a la hora de elegir una plataforma y así comenzar a trabajar con la tecnología que se está ocupando, es decir, BLE Mesh. Mencionando de manera breve lo que se dice en dicho documento, Bluetooth SIG manifiesta, en primer lugar, que para garantizar el éxito del producto es importante seleccionar la arquitectura correcta, ya sea un *single chip*, en la que un solo módulo o chip hace frente a todas las tareas y requisitos del sistema, o un *dual chip*, que utiliza un microcontrolador más un coprocesador de malla Bluetooth. Una vez seleccionado se puede trabajar con ese proveedor de silicio para obtener el datasheet, el entorno de desarrollo recomendado, el código fuente de muestra, etc. En segundo lugar, es necesario estimar el impacto del consumo de memoria en función de la configuración de la red en malla que se quiere diseñar. Y, por último, hay que considerar los factores claves para reducir el consumo de energía en un producto Bluetooth Mesh. Este último aspecto debe estar motivado por la necesidad de ejecutar aplicaciones el mayor tiempo posible mientras se consume una energía mínima, especialmente en sistemas con restricciones de energía.

Algunas famosas compañías como *Silicon Labs* [21], *Nordic Semicondictor* [22], *STMicroelectronics* [23] y *Espressif Systems* [24], han lanzado de manera exclusiva kits de desarrollo software (SDK) para trabajar con Bluetooth Mesh.

El SDK para el desarrollo de Bluetooth en malla que ofrece *Silicon Labs*, es descargable desde el sitio web de la compañía [28]. Para poder comenzar a trabajar con esta tecnología inalámbrica, se requiere el uso completo del entorno de desarrollo integrado *Simplicity Studio* [29], el cual incluye demostraciones precompiladas, notas de aplicación y ejemplos.

[21] https://www.silabs.com/
[22] https://www.nordicsemi.com/
[23] https://www.st.com/content/st_com/en.html
[24] https://www.espressif.com/

Además, *Silicon Labs* tiene una aplicación disponible en Android [30] e iOS [31] llamada *Bluetooth Mesh by Silicon Labs*, para aprovisionar, configurar y controlar los nodos Bluetooth Mesh. Por otro lado, la compañía recomienda comprar tres o cuatro kits *EFR32xG21 Bluetooth Starte* [32] para comenzar a construir un prototipo de una red en malla. Estos kits se basan en el SoC de bajo consumo *EFR32xG21*, y tienen un precio de aproximadamente 84€ a día de hoy.

Por su parte, *Nordic Semiconductor* ofrece una solución completa para el desarrollo de Bluetooth Mesh. En primer lugar, agregó a su línea de kits de desarrollo el *nRF5 SDK for Mesh* [33], que incluye la pila de perfil de malla Bluetooth, bibliotecas, ejemplos, instrucciones claras sobre cómo construir una red o cómo crear nuevos modelos y, además, incluye las herramientas *CMake y SEGGER Embedded Studio.* En segundo lugar, la empresa también ofrece la familia de SoC *nRF52*, que cuenta con diferentes tamaños y capacidades de memoria, lo que permite al desarrollador seleccionar el SoC perfecto para su aplicación. Por ejemplo, se puede usar el kit hardware *nRF52 DK* [34], que facilita el desarrollo de soluciones BLE Mesh, pues cuenta con el SoC *nRF52832* y su precio está en torno a los 35€. Por último, *Nordic Semiconductor* ofrece una aplicación móvil tanto para Android [35] como para iOS [36]denominada *nRF Mesh*. Con ella se puede aprovisionar, configurar y controlar fácilmente las redes de malla Bluetooth.

STMicroelectronics ha adoptado un enfoque similar al de *Silicon Labs* y *Nordic Semiconductor*, pues ofrece el SDK *BlueNRG-Mesh* [37], que permite el desarrollo de soluciones BLE Mesh basadas en los SoC de bajo consumo *BlueNRG-2*. La placa *STEVAL-IDB008V2* [38] de esta compañía incorpora este SoC y tiene un valor de 61€. Además, *STMicroelectronics* también cuenta con una aplicación *ST BLE Mesh* tanto para móviles Android [39] como para iOS [40].

Por último, el equipo Bluetooth Low Energy de *Espressif* desarrolló a principios del año 2019 el SDK *ESP-BLE-MESH* [41]. Tras más de un año de investigación, la implementación de este SDK, además de admitir crear aplicaciones relacionadas con Bluetooth en malla, pues proporciona un entorno de desarrollo muy completo llamado ESP-IDF, también tiene la certificación oficial[25] de Bluetooth SIG. Para trabajar con él, la compañía ofrece el hardware basado en el SoC ESP32. Este SoC está integrado en placas como la *ESP32-DevKitC* [42], cuyo precio oscila entre los 5€ y los 10€ (dependiendo del proveedor) y las cuales están pensadas para la creación rápida de prototipos.

Obviamente, trabajar con cualquiera de estas compañías sería positivo para empezar a desarrollar aplicaciones Bluetooth Mesh, ya que, como se ha visto, todas ellas proporcionan un entorno de desarrollo en el que crear dichas aplicaciones y también ofrecen el hardware que soporta esta nueva tecnología de red. Como en este trabajo se quiere iniciar al lector a BLE Mesh y, por tanto, no se pretende crear una aplicación de alta complejidad, sino realizar pruebas de concepto útiles para fijar los conceptos de la red en malla Bluetooth, se seleccionan las herramientas que ofrece *Espressif*, puesto que las placas de desarrollo que

[25] https://launchstudio.bluetooth.com/ListingDetails/76255

se proporcionaron fueron las de esta empresa debido, en primer lugar, a su precio y, en segundo lugar, a su disponibilidad.

Por otro lado, y antes de profundizar sobre el hardware y software seleccionado, es importante destacar que hay proyectos de código abierto, como, por ejemplo, *Zephyr Project*[26], que incluyen funcionalidad de BLE Mesh para que los desarrolladores puedan implementarlo en sus propios proyectos. De hecho, el SDK *ESP-BLE-MESH* de *Espressif* utilizado en el presente trabajo está construido sobre el SDK de *Zephyr*.

Zephyr [43] es un sistema operativo escalable y de tiempo real (RTOS) para dispositivos IoT con recursos limitados de múltiples arquitecturas. Se trata de un proyecto colaborativo, organizado por *The Linux Foundation*[27], en el que participan empresas de primer nivel con la intención de reducir la fragmentación para desarrolladores de soluciones IoT y de proporcionar una solución que resuelva los problemas de desarrollo en el ámbito integrado como la seguridad o la conectividad. Al ser una comunidad colaborativa, el proyecto *Zephyr* está en constante crecimiento, ya que los que trabajan en el espacio de desarrollo integrado fomentan las comunicaciones y el intercambio de conocimientos para mejorar continuamente el proyecto.

Zephyr Project, como se comentaba anteriormente, da funcionalidad a Bluetooth Mesh, pues cubre todas las capas de su pila. Para hacer funcionar esta tecnología, *Zephyr* pone como únicos requisitos que se use QEMU[28] con BlueZ[29] ejecutándose en host (véase apartado C del Apéndice III) o que se use una placa con soporte Bluetooth LE, ya que BLE Mesh puede implementarse en cualquier dispositivo compatible con Bluetooth 4.0 o posterior. Los desarrolladores del proyecto ofrecen en la página oficial [44] una gran variedad de placas a las que dan soporte, entre ellas están las placas basadas en el chip ESP32, las cuales fueron seleccionadas antes para poder crear una solución Bluetooth Mesh con ellas.

En primera instancia, se seleccionó *Zephyr* como herramienta base para desarrollar este proyecto final de máster al considerarse un sistema operativo bastante novedoso para incluso reemplazar a nivel, por ejemplo, docente a otros sistemas operativos, ya sea por la gran estabilidad y calidad de uso con la que lo describen, la gran cantidad de documentación que presenta, los ejemplos ofrecidos, las diversas funcionalidades, etc. Sin embargo, tras las pertinentes instalaciones y pruebas se comprobó que a pesar de que el soporte de *Zephyr Project* existe para ESP32, aún es muy básico y no funciona para crear despliegues serios. Por ello, tal vez sería mejor idea comenzar con placas que tengan sus propias muestras específicas como las *BBC micro:bit*[30], las cuales la propia organización de

[26] https://www.zephyrproject.org/

[27] Consorcio tecnológico sin ánimo de lucro establecido para adoptar el crecimiento de Linux (https://www.linuxfoundation.org/).

[28] Software gratuito y de código abierto que recrea una máquina virtual dentro de un sistema operativo. Su objetivo es emular un sistema operativo dentro de otro, contando el emulado con su propio procesador y memoria.

[29] Pila oficial de Bluetooth de Linux. Proporciona soporte para las capas y protocolos centrales de Bluetooth.

[30] https://microbit.org/

Bluetooth SIG propone usarlas junto a *Zephyr* para adentrar a los desarrolladores de software en la creación de aplicaciones BLE Mesh a través de su guía de estudio [45].

4.1. Hardware seleccionado

Como se comentaba, se ha seleccionado el *System on Chip* ESP32 de *Espressif* como parte hardware del proyecto. Este SoC es capaz de realizar tanto tareas de cómputo como de comunicación, ya que proporciona distintas alternativas para hacerlo. Integra características como Wi-Fi y Bluetooth, y, además es capaz de ejecutar aplicaciones en tiempo real, lo cual le hace ser un dispositivo muy interesante y con evidencias de que fue especialmente diseñado para el Internet de las Cosas.

Para conocer un poco más de las características técnicas que puede aportar el ESP32, se puede visualizar la Figura 14, que muestra el diagrama de bloques de este SoC.

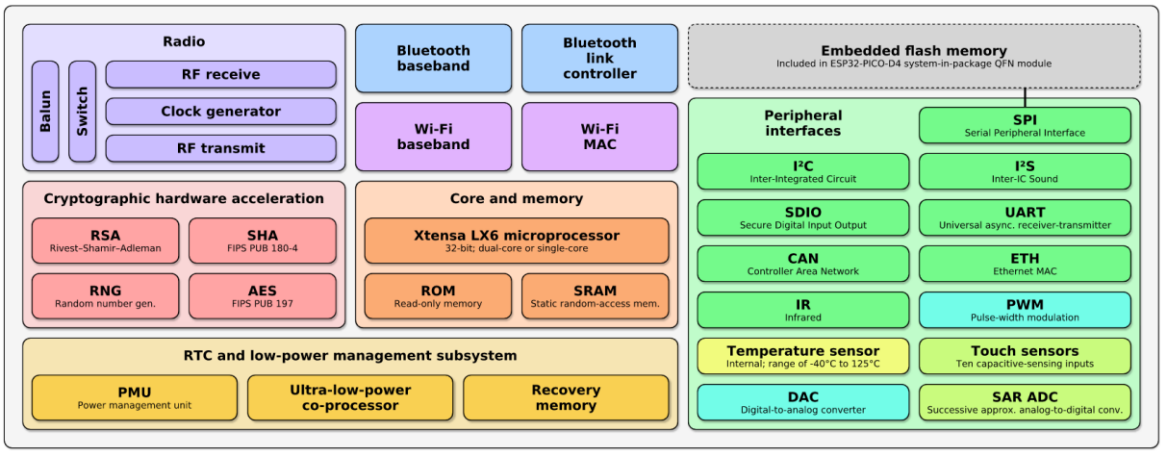

Figura 14. Diagrama de bloques del SoC ESP32 [46]

Algunas de las características generales a destacar de este chip serían:

- Puede contar con uno o dos microprocesadores Xtensa LX6 de 32 bits, con una frecuencia de reloj de hasta 240 MHz y un rendimiento de hasta 600 DMIPS.
- Memoria SRAM de 520 KB para instrucciones y datos.
- Memoria ROM de 448 KB para funciones de núcleo y *boot*.
- Memoria flash de 4 MB, ampliable hasta 16 MB.
- Soporte para Wi-Fi 802.11 b/g/n (802.11n @ 2.4 GHz hasta 150 Mbit/s).
- Soporte para Bluetooth v4.2 BR/EDR y BLE.
- Soporte de protocolos de comunicación I²C, I²S, SPI, UART, ETH, …
- Soporte de características de seguridad del estándar IEEE 802.11, incluyendo WFA, WPA/WPA2 y WAPI.
- Regulador interno para administrar la energía.

Por otro lado, es importante diferenciar entre chip, módulo y placa de desarrollo. El módulo, en este caso, llevará el chip ESP32 integrado junto con un oscilador de cristal y un circuito de adaptación de antena, mientras que la placa de desarrollo integrará el módulo en una PCB que permite conexión serie y alimentación por USB, y cuenta con los botones *boot* y *reset*, y pines soldados a la misma.

Existen diversas placas de desarrollo basadas en el SoC ESP32, sin embargo, la seleccionada es la *ESP32-DevKitC*, en concreto la versión 4 de esta (*ESP32-DevKitC V4* [47]). El módulo con el que cuenta es el *ESP-WROOM-32*, que por defecto tiene 4MB de memoria flash, aunque es ampliable a 8MB o 16MB. Su interfaz de programación es a través del puerto serie USB, el cual también le proporciona la alimentación de 5V a la placa. Finalmente, en la Figura 15 se puede observar la distribución de pines en la placa de desarrollo *ESP32-DevKitC V4*.

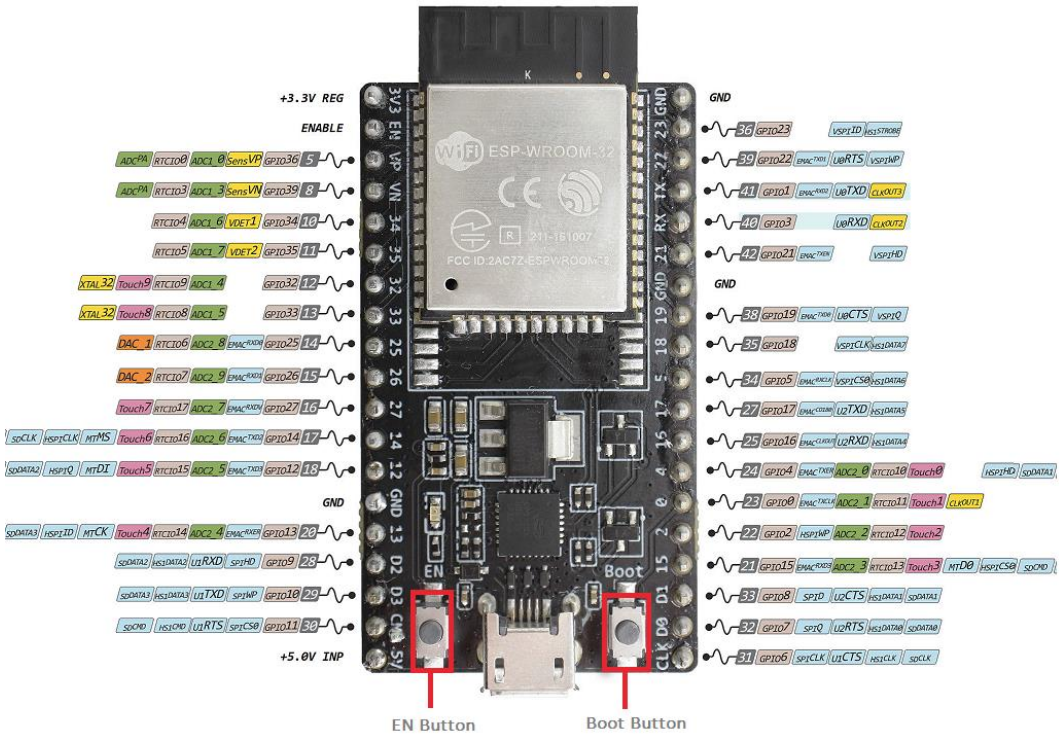

Figura 15. Pinout del ESP32 DevKit v4 [48]

4.2. Software seleccionado

El hardware es solo una parte de un sistema IoT, por ello, la empresa *Espressif Systems*, además de diseñar el SoC ESP32, creó un entorno de desarrollo, basado en línea de comandos, denominado *Espressif IoT Development Framework* (ESP-IDF), para poder programar aplicaciones de manera nativa en estos chips en lenguaje C/C++.

Como ESP32 tiene gran popularidad, la comunidad ha migrado el ESP-IDF a MicroPython IDE (*MicroPython Integrated Development Enviroment*) y a Arduino IDE (*Arduino Integrated Development Enviroment*) como alternativas para el desarrollo de aplicaciones con el ESP32. Por ejemplo, el entorno de desarrollo de Arduino es uno de los más utilizados debido a su facilidad de uso. A pesar de permitir compilar una aplicación, flashearla en la tarjeta de desarrollo y monitorear los mensajes de esta, no tiene capacidades de depuración y carece de soporte para múltiples archivos en un proyecto. La alternativa a esto sería usar Eclipse IDE [49], que es un entorno de código abierto, que integra ESP-IDF y soporta depuración *JTAG*, *OpenOCD* y *GDB*.

A pesar de que Eclipse IDE es más visual, ya que los proyectos basados en ESP-IDF se desarrollan desde una terminal y esto puede resultar más complejo, se determina que, si

se quiere desarrollar sistemas embebidos más sofisticados y optimizados, lo mejor es elegir ESP-IDF, pues además de basarse en el código nativo de FreeRTOS para el desarrollo de sus funciones, tiene mayor número de recursos al poseer el soporte del fabricante. Además, la funcionalidad de ESP-IDF incluye obviamente la compilación y descarga del firmware sobre las placas que integran el ESP32 y la configuración basada en menú, la cual permite configurar distintos aspectos del dispositivo, como, por ejemplo, la tabla de particiones, la configuración del Wi-Fi y contraseña, etc.

Este framework, además de ser de código abierto y de estar disponible en GitHub [50], se puede usar desde cualquier sistema operativo, ya sea, Windows, Linux o macOS. En este caso, para realizar el estudio, se ha configurado la versión 4.2 del entorno de desarrollo en Ubuntu, una distribución de Linux basada en la arquitectura Debian. Para obtener más información de cómo hacer esto posible se puede consultar el Apéndice II.

Tras haber visto tanto la parte hardware como la software, se puede finalizar esta sección con una pequeña lista que reúna lo que se necesitará para poder desarrollar el proyecto:

- Un PC con cualquier sistema operativo de los antes mencionados (Windows, Linux o macOS).
- Una o varias placas que integren el SoC ESP32 junto con cable USB A/micro USB B.
- Editor de texto para escribir los programas de los proyectos en C (*Sublime*[31]).
- ESP-IDF, pues contiene la API (biblioteca de software y código fuente) para ESP32.
- Cadena de herramientas para compilar el código de la ESP32.
- Herramientas de construcción como *CMake* y *Ninja* para construir una aplicación completa para la propia ESP32.

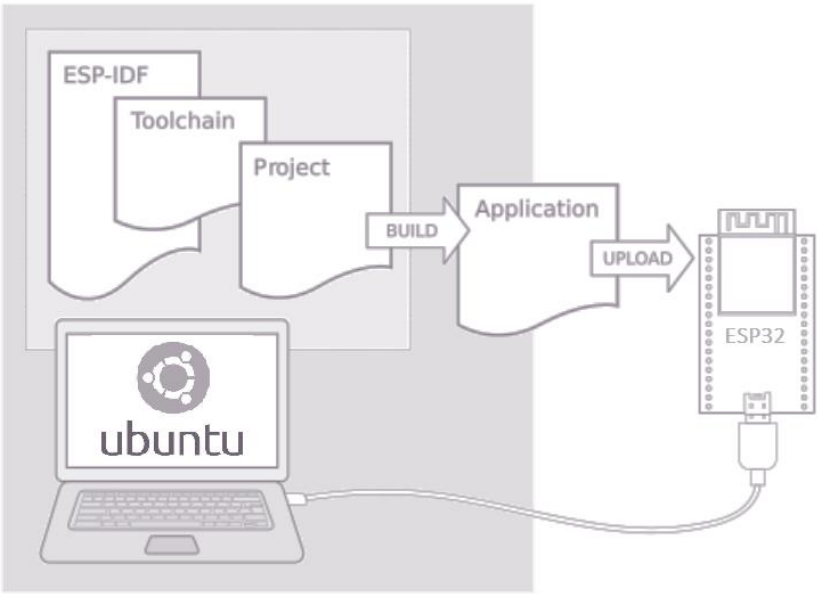

Figura 16. Esquema de la creación de aplicaciones para las ESP32 [51]

[31] https://www.sublimetext.com/

Capítulo 5

Aplicaciones IoT

Los sistemas de comunicaciones inalámbricas basados en topologías de red en malla han demostrado ofrecer un enfoque eficaz para proporcionar, entre otras cosas, cobertura en grandes áreas. En cambio, hasta ahora se han basado en tecnologías de nicho, incompatibles con la mayoría de los smartphones, ordenadores, y dispositivos inteligentes de los consumidores [17]. Por ejemplo, como se vio en el capítulo 2, Zigbee es una tecnología dentro del mercado IoT de *Smart Home*, sin embargo, el problema que presenta es que, si los dispositivos IoT usan este protocolo, los teléfonos inteligentes no pueden conectarse directamente con ellos, puesto que no tienen interfaces inalámbricas Zigbee incorporadas. Por tanto, esto podría considerarse una razón clave del éxito de la tecnología Bluetooth en el mercado IoT, ya que tiene compatibilidad con productos actuales como televisiones inteligentes, ordenadores, teléfonos móviles, etc., y, además, Bluetooth Low Energy con su característica de baja energía, ya hizo de esta tecnología una gran competidora dentro de las soluciones IoT. Ahora, con la llegada del Bluetooth Mesh, se pueden construir redes de dispositivos a gran escala con una topología de red en malla, y, por lo tanto, las redes de gran alcance abren nuevos modelos de uso para la tecnología, particularmente en el campo del IoT.

Hay una amplia gama de sectores donde se puede adoptar Bluetooth Mesh [52]. Actualmente se puede implementar en la industria inteligente (*Smart Industry*), los hogares inteligentes (*Smart Home*), la automatización completa de edificios (*Smart Building*) y otras soluciones IoT donde decenas, cientos o miles de dispositivos necesitan comunicarse entre sí de manera confiable y segura.

En primer lugar, dentro de la industria inteligente, la malla Bluetooth puede ayudar a los fabricantes a alcanzar nuevos niveles de productividad y seguridad a través de aplicaciones como el seguimiento de activos (*asset tracking*), pues su objetivo es monitorear la ubicación de activos específicos que son importantes para un determinado negocio. Existen diversas formas en las que este sector puede aprovechar los servicios de ubicación para mejorar la eficiencia, generar ingresos e impulsar la productividad [53]. Cuanto más grande es una instalación, más importante es realizar un seguimiento de los activos críticos. Por ejemplo, un artículo extraviado en una fábrica o almacén puede costar un tiempo y un esfuerzo valioso que impactan directamente en los resultados de la empresa. Por tanto, las soluciones habilitadas por Bluetooth Mesh pueden ayudar al personal a encontrar activos importantes [54], desde dispositivos médicos y montacargas hasta pacientes hospitalarios y palés de mercancías. Por otro lado, el seguimiento del personal puede mejorar la eficiencia operativa, asegurando que los problemas de mantenimiento o de custodia se resuelvan de manera eficiente. Por último, dentro del IoT industrial también sería interesante destacar aplicaciones en las que se necesita desplegar redes de sensores BLE, que permiten la monitorización en tiempo real del rendimiento y

las tolerancias del sistema, creando máquinas inteligentes y líneas de fabricación completas que implementan un mantenimiento predictivo para resolver rápidamente fallos o errores.

La malla Bluetooth también ha dado soluciones al sector del hogar inteligente, pues ofrece cobertura completa, segura y confiable. Además, está totalmente automatizado, ya que se tiene el control automático de luces, termostatos, detectores de humo, cámaras, timbres, cerraduras, etc. Sin embargo, desde el principio, Bluetooth Mesh fue diseñada para el mercado de los edificios inteligentes [55], específicamente en aplicaciones orientadas a la iluminación comercial. La creciente demanda de los sistemas de control de iluminación a gran escala ha servido como un caso clave para impulsar el aumento de las implementaciones de la malla Bluetooth. Una infraestructura de iluminación inteligente puede ayudar a lograr valiosos ahorros de energía, reducir los costos de mantenimiento y mejorar los servicios para los administradores de edificios. Esta misma infraestructura también puede ayudar a industrias como los hospitales y el transporte a administrar y ubicar activos críticos, así como ayudar a los administradores de edificios a comprender la utilización del espacio para ayudar a priorizar los requisitos de iluminación futuros e incluso las necesidades actuales de limpieza y servicio.

Por otro lado, todos los productos o aplicaciones que incorporan diseños Bluetooth, en este caso, Bluetooth Mesh, deben pasar por lo que se conoce como *Qualification Process* [56]. Este proceso de certificación garantiza que todos los diseños cumplen con la interoperabilidad global y refuerza la marca Bluetooth en beneficio de todos los miembros de Bluetooth SIG. De este modo, la calificación ayuda a las empresas miembro a garantizar que sus productos Bluetooth cumplen con el Acuerdo de licencia de Bluetooth[32] y, por supuesto, con las especificaciones de la malla Bluetooth. Además, todos los productos Bluetooth que completan con éxito el proceso de certificación se añaden a una base de datos de productos Bluetooth [57]. Tras esto, todos los dispositivos y/o aplicaciones se someten a una serie de pruebas para demostrar que efectivamente cumplen con todos los requisitos de la calificación.

En su primer año de vida, la malla Bluetooth consiguió 65 productos calificados, y dos años después presenta un total de 752 productos Bluetooth calificados con capacidad de red en malla [58]. En comparación con otras tecnologías de malla alternativas lanzadas años antes que Bluetooth Mesh, se determina que Bluetooth se convertirá en la tecnología líder en redes *mesh* debido a su crecimiento vertiginoso durante estos años. La ventaja que presenta respecto a las demás tecnologías es que cuenta con las 120 empresas miembro de Bluetooth SIG que participan activamente en el soporte de las redes en malla Bluetooth. Esto es significativamente más de lo que suele ser el caso y es representativo de la demanda de un estándar industrial global para una capacidad de red en malla.

Si se profundiza en la lista de certificaciones, durante los últimos años se observa que los productos como pilas software Bluetooth y los módulos que incluyen capacidad de red en malla fueron los primeros clasificados. Estos productos fueron de compañías como *Espressif, Nordic Semiconductor, Silicon Labs, Qualcomm,* etc. Sin embargo, otro tipo de producto que impulsó el aumento de las implementaciones Bluetooth Mesh fue el

[32] *https://www.bluetooth.com/about-us/governing-documents/*

relacionado con la iluminación y el control de esta. Por ejemplo, la empresa *LEDVANCE*[33] desarrolló los primeros productos de iluminación LED del mundo con certificación[34] de Bluetooth Mesh, pues utilizó la malla Bluetooth para garantizar que varios interruptores de iluminación, sensores y software se fabricasen con estándares universales y se pudiesen usar con unos y otros [59].

A día de hoy, hay múltiples proyectos donde se pueden ver implementaciones de Bluetooth Mesh, desde pequeñas empresas piloto hasta despliegues comerciales a gran escala. Para dar una idea de lo versátiles que pueden ser las aplicaciones de *Smart Lighting*, a continuación, se listan una serie de ejemplos para estar un poco más cerca de la realidad de esta tecnología [60].

- En la oficina de Macq Mobility Management Solutions [61] de Bruselas, se hizo una renovación completa de la iluminación durante un solo fin de semana. La empresa se actualizó a LED e implementó una red de control de iluminación de Bluetooth Mesh de 360 nodos. El sistema de control fue impulsado por sensores con detección de ocupación y recolección de luz natural.

- La Torre Stratosphere [62] de las Vegas es una de las atracciones más emblemáticas de EEUU. Los gerentes del edificio decidieron modernizar la iluminación de su observatorio situado en el piso 108 para crear escenas de iluminación independientes sin tener que cablear un sistema de control. Como era de esperar, los controles de iluminación de Bluetooth Mesh cumplieron estos requisitos.

- En la oficina de OSRAM [63] en Garching (Alemania), se utilizó HubSense, un sistema basado en la malla Bluetooth para actualizar la iluminación de su oficina. En la modernización no hubo necesidad de hacer perforaciones ni añadir cables. Además, las estrategias de control implementadas en la oficina incluyeron detección de ocupación, aprovechamiento de la luz del día y control manual.

- El almacén de motores de Yamaha Motor Corporation [64] en Pleasant Prairie (EEUU) realizó una actualización sin ningún problema de luminarias completas junto con un sistema de control eficiente y consciente de la ocupación. La conectividad de la malla Bluetooth se proporcionó a través de sensores de ocupación inteligentes, pues se desplegaron un total de 320 dispositivos de este tipo por las estanterías del almacén.

Dicho esto, es cierto que todavía existen muchos conceptos erróneos sobre el estado actual del estándar de malla Bluetooth y sus capacidades. La industria de la iluminación ha estado persiguiendo el sueño de IoT durante aproximadamente una década, y, además, la parte de control de iluminación resultó ser más difícil de lo esperado, ya que la industria tardó años en encontrar una solución que cumpliera con los requisitos de iluminación profesional. Después de todo este tiempo, algunas personas o empresas pueden tener dificultades para creer que la espera ha terminado y que se puede lograr una confiabilidad

[33] https://www.ledvance.es/
[34] https://launchstudio.bluetooth.com/ListingDetails/52584

similar a la de un cable en el control inalámbrico. La mejor manera de ponerse al día con Bluetooth Mesh es verlo en acción, por ello, en la siguiente sección se desarrollan dos pruebas de concepto para adentrar al lector a las funcionalidades de la malla Bluetooth.

Por último, y a modo de conclusión, decir que Bluetooth no busca ser la mejor solución para todas las necesidades de conectividad inalámbrica, pero en caso de que un desarrollador, que está creando una solución, necesita transmitir audio entre dos dispositivos, necesita transferir datos entre dos dispositivos, necesita una solución de servicios de ubicación o una solución en la que necesita tener decenas, cientos o incluso miles de dispositivos que se comunican entre sí en una red de malla segura y confiable, entonces Bluetooth se esfuerza por ser la mejor tecnología disponible.

Capítulo 6

Pruebas de concepto

A continuación, el desarrollo de dos pruebas de concepto (*proof of concept*, PoC), permitirá el testeo de la tecnología de red con la que se está trabajando. Con estas pruebas, además de acercar al lector a la implementación software de posibles aplicaciones BLE Mesh, se conseguirá evaluar si el framework ESP-IDF realmente soporta Bluetooth Mesh.

Por otra parte, tanto los códigos implementados en ambas pruebas de concepto como los que se utilizan para desarrollar las diferentes soluciones de aprovisionamiento del Apéndice III, se recogen en el GitHub *https://github.com/lidfer/TFM_BleMesh* .

6.1. Desarrollo de una solución de control de luz RGB con BLE Mesh en ESP-IDF

En esta primera prueba de concepto, se va a describir el proceso de creación de una aplicación de control de luz RGB. Para ello, se usará el modelo *Generic On/Off* de Bluetooth Mesh, el cual está implementado en un ejemplo proporcionado por ESP-IDF. Dentro del GitHub oficial [50] de este entorno de desarrollo, el ejemplo mencionado está disponible en el directorio *esp-idf/examples/bluetooth/esp_ble_mesh/ble_mesh_node/onoff_server*.

6.1.1. Modelo Genérico On/Off

El modelo *Generic On/Off* es el ideal para iniciarse con Bluetooth Mesh, pues al estar dentro de la colección de modelos *generics*, puede ser utilizado por cualquier tipo de dispositivo, ya que, por ejemplo, este modelo en cuestión ofrece un conjunto de capacidades (encendido y apagado de dispositivos) aplicables a cualquier tipo de aplicación.

En este caso, el modelo que se implementa es un modelo servidor que contiene un único estado, el estado *generic onoff*. Se trata de un estado booleano que indica cuando un elemento está encendido, a través de un valor de *0x01,* o cuando está apagado, con un valor de *0x00.* Por su parte, el modelo cliente se encarga de mandar mensajes *Set* o *Get* al servidor para modificar el estado de los nodos o solicitar el estado actual de estos. Sin embargo, en esta ocasión este modelo no se implementa, pues, como se verá más adelante, el smartphone del usuario actuará tanto de cliente como de aprovisionador.

6.1.2. Estructura general de la solución

Antes de proceder a analizar el código de ejemplo que facilita ESP-IDF para familiarizarse con Bluetooth Mesh, se va a hacer un esquema general de la prueba de concepto que se está desarrollando, el cual se muestra en la Figura 17.

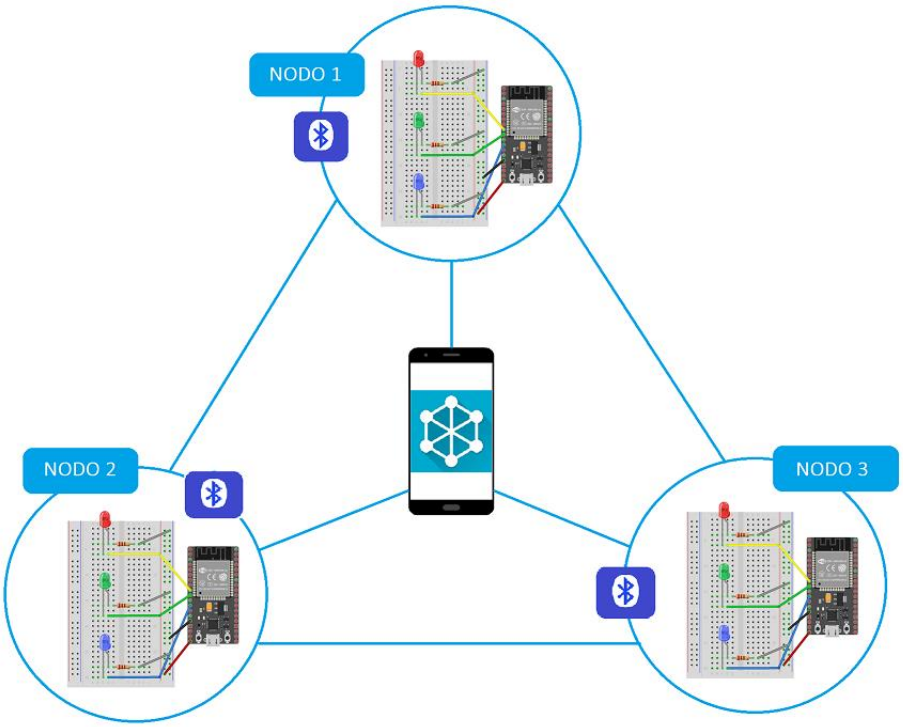

Figura 17. Esquema de la solución de control de luz RGB con BLE Mesh

Como se puede observar, la aplicación de control de luz RGB que se crea en esta prueba de concepto se basa en la implementación de un interruptor de luz usando el modelo *Generic On/Off*, diseñado para controlar hasta 3 lámparas LED RGB a través de una red de malla Bluetooth. Obviamente, el número de lámparas es escalable, solo que a la hora de realizar esta prueba no se cuenta con todo el material deseado, aun así, es suficiente para comprobar el funcionamiento de la red.

En los pines GPIO 25, 26 y 27 de las placas *ESP32-DevKitC V4*, se conectan 3 LEDs, pues las placas de desarrollo simularán distintas lámparas, mientras que dichos LEDS simularán bombillas de diferentes colores. Por tanto, se llega a la conclusión de que las lámparas LED son implementadas por los nodos servidores, mientras que el interruptor se implementa desde el teléfono móvil, es decir, desde el nodo cliente, el cual a su vez actúa inicialmente como administrador de red.

El código que facilita ESP-IDF para, en primer lugar, familiarizar al desarrollador de soluciones Bluetooth Mesh con esta tecnología, y, en segundo lugar, para *flashearlo* sobre las ESP32 que actuarán como nodos servidores, se encuentra en el archivo *main.c* del directorio mencionado anteriormente (*esp-idf/examples/bluetooth/esp_ble_mesh/ble_mesh_node/onoff_server/main*).

En el archivo *main.c*, se comienza con la inicialización tanto de la pila del protocolo Bluetooth Low Energy (con la función *bluetooth_init()*) como de la pila BLE Mesh (con la función *ble_mesh_init()*) (Figura 18).

```c
void app_main(void)
{
    esp_err_t err;

    ESP_LOGI(TAG, "Initializing...");

    board_init();

    err = nvs_flash_init();
    if (err == ESP_ERR_NVS_NO_FREE_PAGES) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK(err);

    err = bluetooth_init();
    if (err) {
        ESP_LOGE(TAG, "esp32_bluetooth_init failed (err %d)", err);
        return;
    }

    ble_mesh_get_dev_uuid(dev_uuid);

    /* Initialize the Bluetooth Mesh Subsystem */
    err = ble_mesh_init();
    if (err) {
        ESP_LOGE(TAG, "Bluetooth mesh init failed (err %d)", err);
    }
}
```

Figura 18. Inicialización de BLE y BLE Mesh

La función *ble_mesh_init()*, que habilita Bluetooth Mesh, se muestra en la figura 19. En ella se registra la función *callback* de aprovisionamiento en la pila BLE Mesh (*esp_ble_mesh_register_prov_callback(example_ble_mesh_provisioning_cb)*), la cual se ejecuta durante el proceso de configuración de la red y permite que la pila de este protocolo genere eventos y notifique a la capa de aplicación sobre procesos importantes de configuración de red. Algunos eventos podrían ser *ESP_BLE_MESH_PROVISION_REG_EVT*, que se genera cuando se completa el proceso de inicialización de BLE Mesh, *ESP_BLE_MESH_NODE_PROV_LINK_OPEN_EVENT*, generado cuando un dispositivo no aprovisionado y un aprovisionador establecen un vínculo, *ESP_BLE_MESH_NODE_PROV_COMPLETE_EVT*, que se envía a la capa de aplicación cuando se completa el aprovisionamiento, etc. Por otro lado, se registran las funciones *esp_ble_mesh_register_config_server_callback* y *esp_ble_mesh_register_generic_server_callback*, que llaman a las funciones *example_ble_mesh_config_server_cb* y *example_ble_mesh_generic_server_cb* respectivamente, que implementan los modelos *Configuration Server* y *Generic Server*. También se utiliza la función *esp_ble_mesh_node_prov_enable(ESP_BLE_MESH_PROV_ADV | ESP_BLE_MESH_PROV_GATT)*, que habilita las funciones de publicidad y escaneo cuando se completa la inicialización de BLE Mesh, es decir, hace que los dispositivos sean visibles para los usuarios para el posterior aprovisionamiento de red. Finalmente, la función *board_led_operation(LED_G, LED_ON)*, se encarga de inicializar el LED RGB.

```c
static esp_err_t ble_mesh_init(void)
{
    esp_err_t err;

    esp_ble_mesh_register_prov_callback(example_ble_mesh_provisioning_cb);
    esp_ble_mesh_register_config_server_callback(example_ble_mesh_config_server_cb);
    esp_ble_mesh_register_generic_server_callback(example_ble_mesh_generic_server_cb);

    err = esp_ble_mesh_init(&provision, &composition);
    if (err) {
        ESP_LOGE(TAG, "Initializing mesh failed (err %d)", err);
        return err;
    }

    esp_ble_mesh_node_prov_enable(ESP_BLE_MESH_PROV_ADV | ESP_BLE_MESH_PROV_GATT);

    ESP_LOGI(TAG, "BLE Mesh Node initialized");

    board_led_operation(LED_G, LED_ON);

    return err;
}
```

Figura 19. Habilitación de BLE Mesh

Llegado este punto, se completa la inicialización y habilitación de Bluetooth Mesh, lo que significa que un usuario ya puede identificar dispositivos para el aprovisionamiento de la red y la posterior transmisión de datos.

A continuación, es el momento de declarar y definir los elementos y modelos necesarios para la solución que se está desarrollando. En este caso, cada nodo contará con tres elementos. El primero será un LED rojo, mientras que los otros dos serán un LED verde y un LED azul. El elemento primario (*primary element*) tendrá dos modelos implementados, mientras que los otros dos elementos solo tendrán un modelo. Esto se debe a que el primario cuenta de manera obligatoria con el modelo *Configuration Server*, que contiene información sobre el nodo, los elementos que contiene y los modelos y características que admite. Por otro lado, en todos los elementos, se encuentra el modelo *Generic On/Off Server*, que implementa la función básica de encender y apagar las luces.

Para empezar, se va a describir cómo se implementaría la estructura de elementos de BLE Mesh. Para ello, habría que hacer la declaración en el fichero *esp_ble_mesh_defs.h* (*esp-idf/components/bt/esp_ble_mesh/api*) de una estructura de elementos la siguiente manera:

```c
typedef struct {
    /** Element Address, assigned during provisioning. */
    uint16_t element_addr;

    /** Location Descriptor (GATT Bluetooth Namespace Descriptors) */
    const uint16_t location;

    const uint8_t sig_model_count;      /*!< SIG Model count */
    const uint8_t vnd_model_count;      /*!< Vendor Model count */

    esp_ble_mesh_model_t *sig_models;   /*!< SIG Models */
    esp_ble_mesh_model_t *vnd_models;   /*!< Vendor Models */
} esp_ble_mesh_elem_t;
```

Figura 20. Descripción de la estructura de un elemento en BLE Mesh

Por su parte, en el archivo principal, se definen los elementos como:

```
static esp_ble_mesh_elem_t elements[] = {
    ESP_BLE_MESH_ELEMENT(0, root_models, ESP_BLE_MESH_MODEL_NONE),
    ESP_BLE_MESH_ELEMENT(0, extend_model_0, ESP_BLE_MESH_MODEL_NONE),
    ESP_BLE_MESH_ELEMENT(0, extend_model_1, ESP_BLE_MESH_MODEL_NONE),
};
```

Figura 21. Definición de los elementos

La macro *ESP_BLE_MESH_ELEMENT* es la que ayuda a definir un elemento BLE Mesh dentro de un nodo. En este caso, se crean tres elementos, siendo el primero de ellos el asociado al LED rojo, es decir, el elemento primario.

Esta macro está definida como se muestra en la Figura 22. Entre las variables de la estructura está *_loc*, que es el primer argumento. Se trata del descriptor de ubicación definido por SIG. En esta ocasión, se establece por defecto el valor *0*. Después está *_mods* y *_vnd_mods*, que indican la matriz del modelo SIG y la matriz de los modelos de proveedores respectivamente. Al último de estos parámetros se le establece la macro *ESP_BLE_MESH_NODE*, ya que en esta ocasión los elementos no cuentan con modelo *Vendor*.

```
#define ESP_BLE_MESH_ELEMENT(_loc, _mods, _vnd_mods)    \
{                                                       \
    .location         = (_loc),                         \
    .sig_model_count  = ARRAY_SIZE(_mods),              \
    .sig_models       = (_mods),                        \
    .vnd_model_count  = ARRAY_SIZE(_vnd_mods),          \
    .vnd_models       = (_vnd_mods),                    \
}
```

Figura 22. Implementación de la macro ESP_BLE_MESH_ELEMENT

Por otra parte, se implementaría la estructura de modelos de BLE Mesh. Para ello, igual que antes, habría que hacer la declaración en *esp_ble_mesh_defs.h* de una estructura de modelos la siguiente manera:

```c
struct esp_ble_mesh_model {
    /** Model ID */
    union {
        const uint16_t model_id;
        struct {
            uint16_t company_id;
            uint16_t model_id;
        } vnd;
    };

    /** Internal information, mainly for persistent storage */
    uint8_t  element_idx;   /*!< Belongs to Nth element */
    uint8_t  model_idx;     /*!< Is the Nth model in the element */
    uint16_t flags;         /*!< Information about what has changed */

    /** The Element to which this Model belongs */
    esp_ble_mesh_elem_t *element;

    /** Model Publication */
    esp_ble_mesh_model_pub_t *const pub;

    /** AppKey List */
    uint16_t keys[CONFIG_BLE_MESH_MODEL_KEY_COUNT];

    /** Subscription List (group or virtual addresses) */
    uint16_t groups[CONFIG_BLE_MESH_MODEL_GROUP_COUNT];

    /** Model operation context */
    esp_ble_mesh_model_op_t *op;

    /** Model-specific user data */
    void *user_data;
};
```

Figura 23. Descripción de la estructura de un modelo en BLE Mesh

A continuación, se define la estructura de los diferentes modelos. Las matrices *root_models[]*, *extend_model_0[]* y *extend_model_1[]* se usan para indicar el número de estructuras de modelos existentes.

```c
static esp_ble_mesh_model_t root_models[] = {
    ESP_BLE_MESH_MODEL_CFG_SRV(&config_server),
    ESP_BLE_MESH_MODEL_GEN_ONOFF_SRV(&onoff_pub_0, &onoff_server_0),
};

static esp_ble_mesh_model_t extend_model_0[] = {
    ESP_BLE_MESH_MODEL_GEN_ONOFF_SRV(&onoff_pub_1, &onoff_server_1),
};

static esp_ble_mesh_model_t extend_model_1[] = {
    ESP_BLE_MESH_MODEL_GEN_ONOFF_SRV(&onoff_pub_2, &onoff_server_2),
};
```

Figura 24. Definición de los modelos

Como se puede observar en la Figura 24, en la primera matriz (*root_models[]*), la cual está asociada al primer elemento, se implementa el modelo obligatorio (*Configuration Server*), a través de la macro *ESP_BLE_MESH_MODEL_CFG_SRV* y un modelo *Generic On/Off Server* a través de la macro *ESP_BLE_MESH_MODEL_GEN_ONOFF_SRV*. Por otro lado, el segundo y tercer elemento solo implementa el modelo *Generic On/Off Server*.

Volviendo a la estructura de la Figura 23, se tiene que dentro de ella se encuentra otra estructura importante. Se trata de *esp_ble_mesh_model_op_t *op*, que apunta a la estructura de operación que define el estado del modelo. La declaración de dicha estructura se muestra en la siguiente figura.

```
typedef struct {
    const uint32_t    opcode;   /*!< Message opcode */
    const size_t      min_len;  /*!< Message minimum length */
    esp_ble_mesh_cb_t param_cb; /*!< Callback used to handle message. Initialized by the stack. */
} esp_ble_mesh_model_op_t;
```

Figura 25. Descripción de la estructura de operación

En la declaración de la estructura de operación hay tres variables. La primera es *opcode*, que es el código de operación asociado a los distintos tipos de mensajes para poder identificarlos. Como se especifica en la documentación de Bluetooth Mesh [21], el código de operación de un modelo SIG debe ser de 1 o 2 bytes. En este caso, al tratase del modelo *Generic On/Off*, se muestra en la Tabla 3 los mensajes junto con sus códigos de operación disponibles para este modelo. La siguiente variable es *min_len*, que marca la longitud mínima de los mensajes recibidos por el estado, y, finalmente, las aplicaciones deben establecer a la variable *param_cb* un valor de *0*.

Server Model	Message Name	Opcode
Generic On/Off	*Generic On/Off Get*	*0x82 0x01*
	Generic On/Off Set	*0x82 0x02*
	Generic On/Off Set Unacknowledged	*0x82 0x03*
	Generic On/Off Status	*0x82 0x04*

Tabla 3. Códigos de operación del modelo Generic On/Off [21]

6.1.3. Aprovisionamiento con la aplicación *nRF Mesh*

Dentro de un despliegue Bluetooth Mesh, el aprovisionador toma un rol muy importante, ya que se encarga de conectar cada uno de los nodos de la red para después configurarlos de tal forma que todos pertenezcan a la misma red BLE Mesh de forma segura.

Durante el desarrollo de este proyecto final, se han estudiado diversas formas de aprovisionar los nodos de una red (véase Apéndice III), sin embargo, en esta prueba de concepto en concreto se usa la aplicación *nRF Mesh* (Figura 26), de la cual ya se habló anteriormente en el capítulo 4. Se selecciona la versión Android [35] de esta aplicación creada por *Nordic Semiconductor* puesto que desde la página oficial del SDK *ESP-BLE-MESH* de *Espressif* se recomienda su uso para demostrar de una manera muy visual cómo aprovisionar dispositivos.

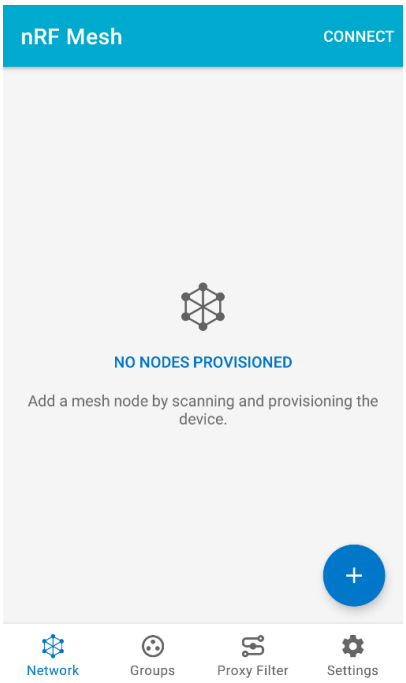

Figura 26. Pantalla de inicio de la aplicación nRF Mesh

Una vez cargado el programa descrito en la sección anterior en los nodos servidores, las placas se inicializan simulando ser lámparas LED RGB. Después, desde la aplicación *nRF Mesh* instalada en el dispositivo móvil del usuario se puede comenzar la búsqueda de nodos no aprovisionados que están dentro de su alcance.

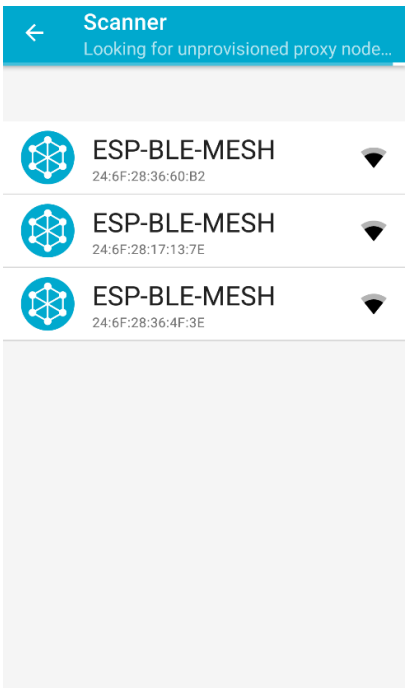

Figura 27. Escaneo de dispositivos no aprovisionados

Como se muestra en la Figura 27, se detectan las tres placas, por tanto, habrá que ir aprovisionando de una en una. Tras seleccionar uno de los tres dispositivos, el teléfono establece con éxito una conexión con él. Cuando el usuario quiera aprovisionar solo debe pulsar *Identify*, que mostrará la lista de capacidades del nodo junto con un nuevo botón denominado *Provision*.

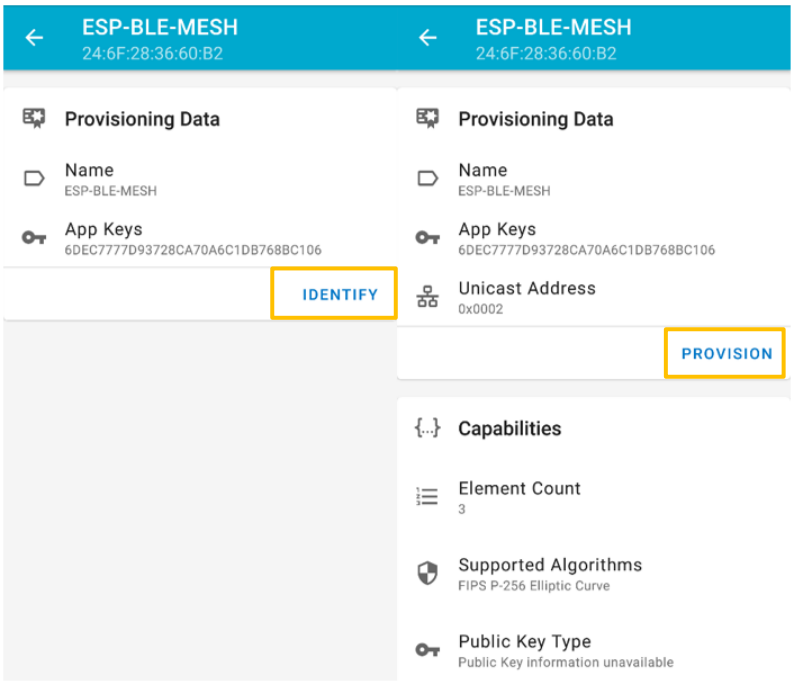

Figura 28. Botones Identify y Provision

Antes de pulsar *Provision*, se puede modificar la dirección *unicast* del dispositivo o comprobar qué prestaciones presenta, por ejemplo, que cuenta con tres elementos diferentes. Tras pulsar *Provision*, se produce un intercambio de mensajes entre el aprovisionador y el dispositivo, y después, de unos pocos segundos, se completa con éxito la incorporación del nodo a la red *mesh*.

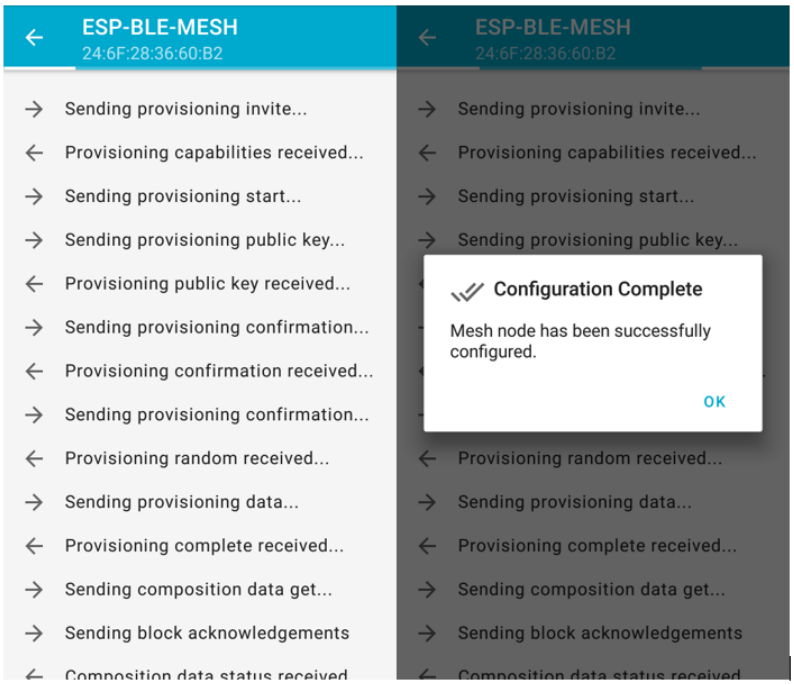

Figura 29. Configuración completada tras pulsar el botón Provision

Desde el monitor serie del dispositivo, se pueden ver signos sobre cómo avanza el proceso de aprovisionamiento de este nodo a través de los distintos eventos que se han generado (Figura 30). Como se sabe, el evento *ESP_BLE_MESH_NODE_PROV_LINK_OPEN_EVENT* indica que el aprovisionador y el dispositivo no aprovisionado han establecido un vínculo. Además, aparece la dirección *unicast* asignada al primer elemento (*0x0002*) y, a través del evento *ESP_BLE_MESH_MODEL_OP_APP_KEY_ADD* se indica cómo al nodo se le ha asociado una *AppKey*.

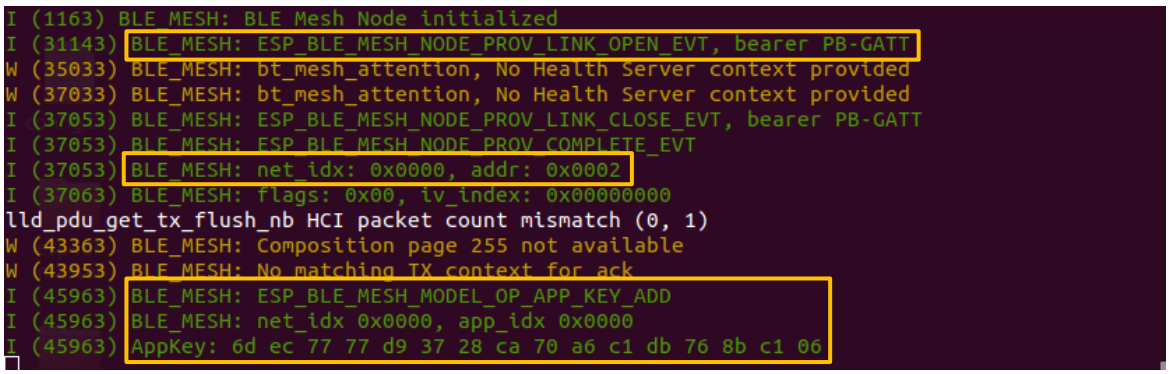

Figura 30. Monitorización del primer nodo tras el aprovisionamiento

Cuando se finalizan los procedimientos con cada uno de los tres dispositivos, se muestra en la pantalla principal de la aplicación una lista con los nodos ya configurados correctamente.

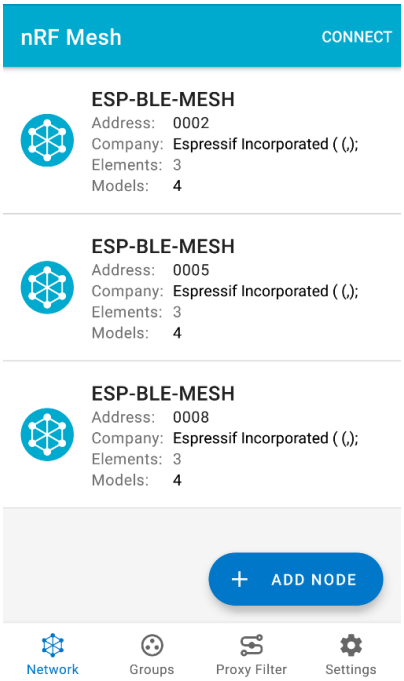

Figura 31. Lista de nodos aprovisionados

A continuación, se debe vincular la *AppKey* con el modelo *Generic On/Off Server* dentro de cada uno de los elementos de los diferentes nodos. Es importante destacar que no es necesario vincular la *AppKey* con el modelo *Configuration Server* perteneciente al elemento primario, ya que solo usa la *DevKey* para cifrar mensajes en la capa *Upper Transport*.

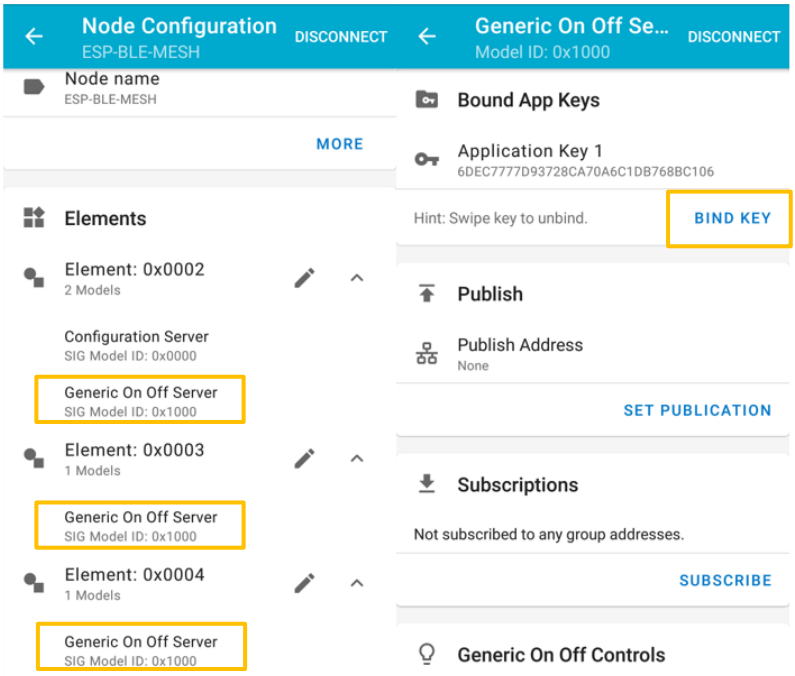

Figura 32. Vinculación de la AppKey

Ahora que todos los modelos *Generic On/Off Server* de los tres elementos están vinculados con la *AppKey* adecuada, se puede controlar el encendido y el apagado de los distintos LEDs dentro de los controles de cada elemento.

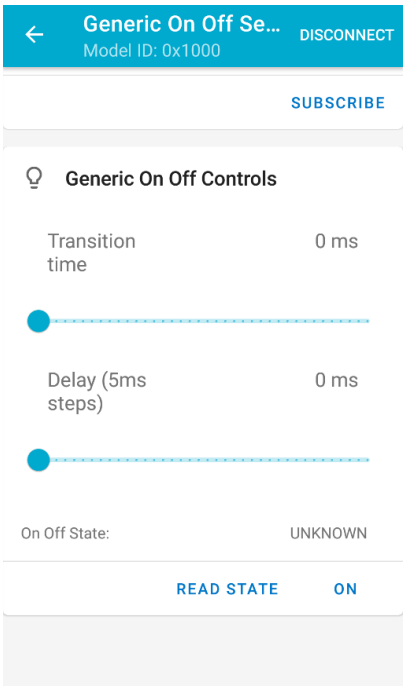

Figura 33. Control on/off desde el elemento

Una vez completado este despliegue BLE Mesh, es posible realizar alguna prueba que demuestre el correcto funcionamiento de este. Un ejemplo de experimento sería el que demostrase una de las ventajas de la malla Bluetooth, en la que el rango de cobertura no se queda en el alcance permitido para los dispositivos Bluetooth Low Energy (aproximadamente 20 metros en interiores), sino que se amplía al usar esta topología de red.

Según la teoría, aunque se dé el caso de que el nodo servidor destino no esté en alcance directo del aprovisionador, el mensaje salta por los nodos intermedios ya aprovisionados hasta llegar al origen. El experimento por tanto consiste en colocar un nodo servidor ya aprovisionado fuera del alcance del cliente, es decir, del teléfono móvil con la aplicación *nRF Mesh*, y ver que los mensajes no llegan (Figura 34).

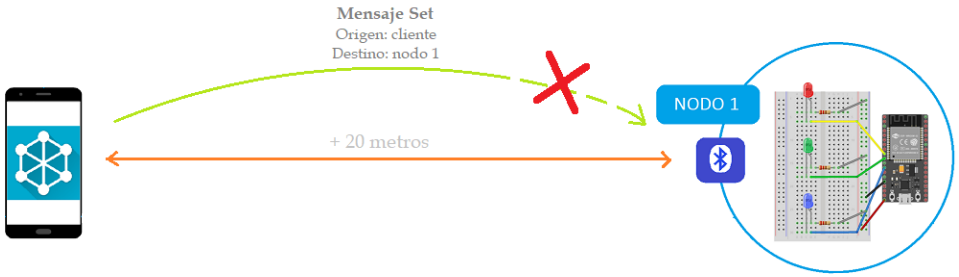

Figura 34. Esquema de la pérdida de mensaje

Después, al colocar un nuevo nodo servidor perteneciente a la misma red en malla a mitad de camino entre ambos, los mensajes comienzan a llegar a ese nodo (Figura 35).

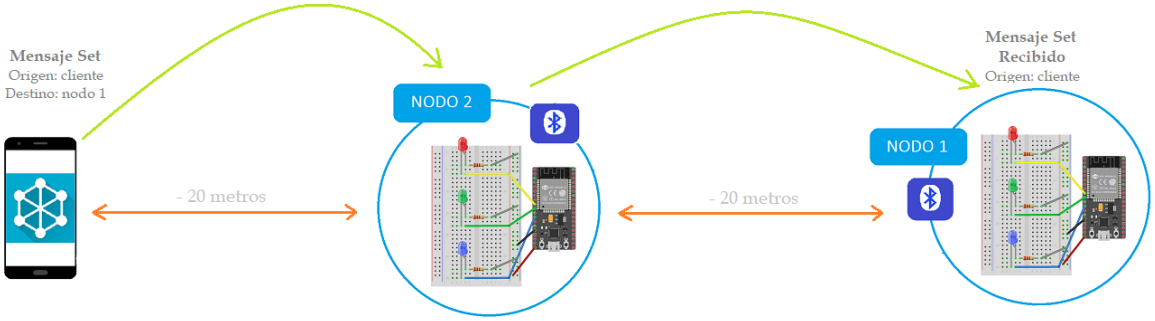

Figura 35. Esquema de la recepción del mensaje

Obviamente, se podría aumentar la distancia entre nodos siempre y cuando haya más nodos por la red que hagan llegar los mensajes al nodo más retirado del origen. Como se ve, esto garantiza que el alcance de la red *mesh* sea superior al área de cobertura que permite una simple conexión Bluetooth LE en topología de estrella. No obstante, se deberá tener en cuenta el TTL que implementa una limitación para controlar el número máximo de "saltos" sobre los que se retransmite un mensaje. Si, por ejemplo, el TTL es 7 y hay más nodos de ese número, es muy probable que el mensaje no llegue a su destino si este es el más alejado del origen y, por lo tanto, el mensaje debe saltar más de 7 veces por distintos nodos de la red para acercarse al destino.

Por otro lado, para usar más conceptos relacionados con Bluetooth Mesh y probar otras funcionalidades de la aplicación *nRF Mesh*, el móvil del usuario ahora se comportará como un interruptor de luz. Para ello, habrá que cambiar a la pestaña *Groups* de la app y después crear tres grupos diferentes como se muestra en la siguiente figura.

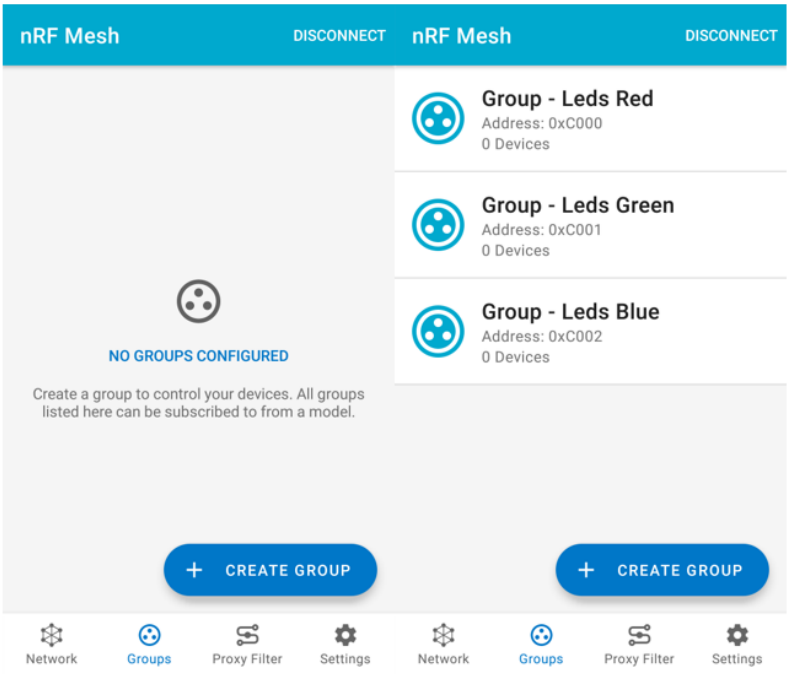

Figura 36. Creación de diferentes grupos en nRF Mesh

Por otra parte, los modelos *Generic On/Off Server* de los elementos primarios de los tres nodos, es decir, el modelo asociado al LED de color rojo, se suscriben a una dirección de grupo (*0xC000*) denominada *Group - Leds Red* (Figura 37).

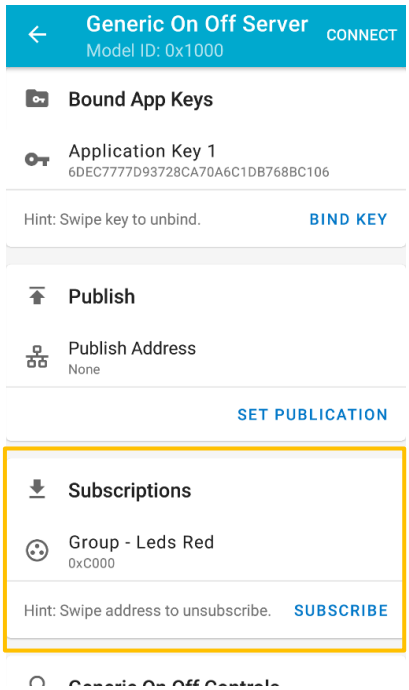

Figura 37. Suscripción del primer elemento al grupo Leds Red

De igual manera, el *modelo Generic On/Off Server* del segundo elemento de los tres nodos, es decir, el modelo que controla el LED de color verde, se va a suscribir a la dirección de grupo *0xC001*, denominada *Group – Leds Green*, mientras que el modelo *Generic On/Off*

Server del tercer elemento de los tres nodos, es decir, el modelo que controla el LED de color azul, se va a suscribir a la dirección de grupo *0xC002* (*Group – Leds Blue*).

Desde el monitor serie de los tres nodos, se puede ver como todos los elementos se han suscrito correctamente a los grupos deseados.

```
I (579903) BLE_MESH: ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD
I (579903) BLE_MESH: elem_addr 0x0002, sub_addr 0xc000, cid 0xffff, mod_id 0x1000
I (595483) BLE_MESH: ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD
I (595483) BLE_MESH: elem_addr 0x0003, sub_addr 0xc001, cid 0xffff, mod_id 0x1000
I (630643) BLE_MESH: ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD
I (630643) BLE_MESH: elem_addr 0x0004, sub_addr 0xc002, cid 0xffff, mod_id 0x1000
```

Figura 38. Elementos del nodo 1 suscritos a sus correspondientes direcciones de grupo

```
I (687312) BLE_MESH: ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD
I (687312) BLE_MESH: elem_addr 0x0005, sub_addr 0xc000, cid 0xffff, mod_id 0x1000
I (698132) BLE_MESH: ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD
I (698132) BLE_MESH: elem_addr 0x0006, sub_addr 0xc001, cid 0xffff, mod_id 0x1000
I (708072) BLE_MESH: ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD
I (708072) BLE_MESH: elem_addr 0x0007, sub_addr 0xc002, cid 0xffff, mod_id 0x1000
```

Figura 39. Elementos del nodo 2 suscritos a sus correspondientes direcciones de grupo

```
I (747793) BLE_MESH: ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD
I (747793) BLE_MESH: elem_addr 0x0008, sub_addr 0xc000, cid 0xffff, mod_id 0x1000
I (758663) BLE_MESH: ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD
I (758663) BLE_MESH: elem_addr 0x0009, sub_addr 0xc001, cid 0xffff, mod_id 0x1000
I (773183) BLE_MESH: ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD
I (773193) BLE_MESH: elem_addr 0x000a, sub_addr 0xc002, cid 0xffff, mod_id 0x1000
```

Figura 40. Elementos del nodo 3 suscritos a sus correspondientes direcciones de grupo

Ahora ya se puede controlar de manera simultánea el encendido o el apagado de los LEDs de todos los nodos que componen la red BLE Mesh que se ha creado. Como se comentó antes, el administrador de red, es decir, el teléfono móvil, actuará de cliente, puesto que a través de un *switch* publicará el estado encender o apagar para un grupo de LEDs determinado. Los modelos, suscritos a ese *switch* en particular, cambiarán su estado dependiendo de lo que el usuario quiera.

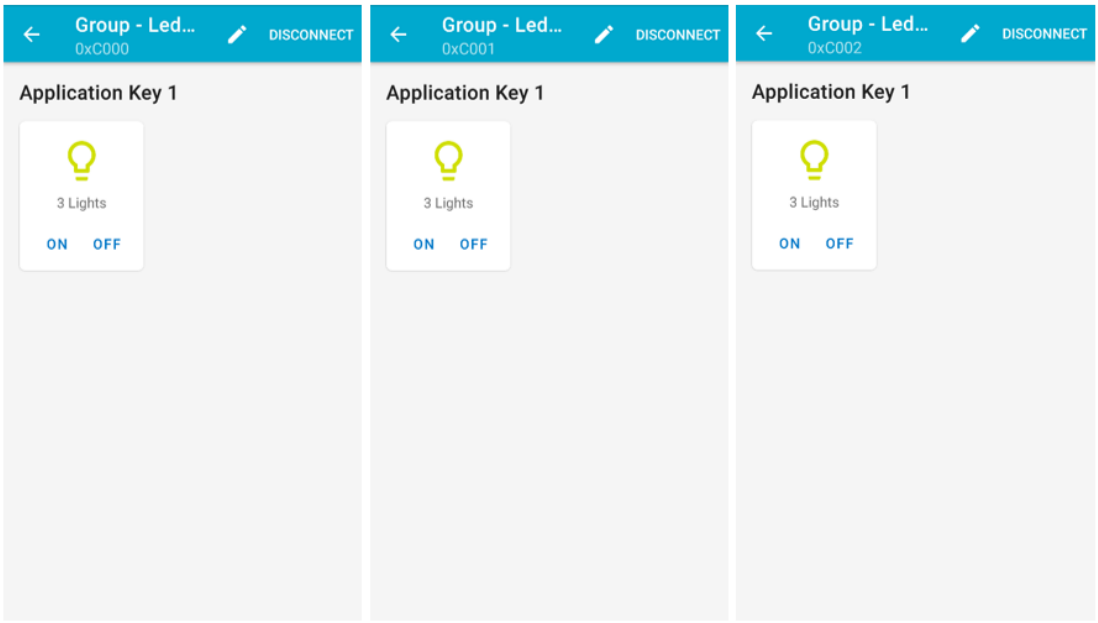

Figura 41. Interruptores administrados por el cliente

Al pulsar *ON*, en los monitores serie de las placas llega el evento que informa del cambio de estado en el modelo *Generic On/Off Server* de los tres nodos, ya que se ha producido el encendido de los LEDs rojos. Después, si se pulsan dos veces seguidas el botón *OFF*, se vuelve a enviar un evento de cambio de estado, sin embargo, nos devuelve un *warning* informando de que los LEDs rojos ya están apagados.

```
I (947233) BLE_MESH: event 0x00, opcode 0x8203, src 0x0001, dst 0xc000
I (947233) BLE_MESH: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (947233) BLE_MESH: onoff 0x01
I (950533) BLE_MESH: event 0x00, opcode 0x8203, src 0x0001, dst 0xc000
I (950533) BLE_MESH: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (950533) BLE_MESH: onoff 0x00
I (982173) BLE_MESH: event 0x00, opcode 0x8203, src 0x0001, dst 0xc000
I (982173) BLE_MESH: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (982173) BLE_MESH: onoff 0x00
W (982173) BOARD: led red is already off
```

Figura 42. Comportamiento del nodo 1

```
I (948802) BLE_MESH: event 0x00, opcode 0x8203, src 0x0001, dst 0xc000
I (948812) BLE_MESH: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (948812) BLE_MESH: onoff 0x01
I (952122) BLE_MESH: event 0x00, opcode 0x8203, src 0x0001, dst 0xc000
I (952122) BLE_MESH: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (952132) BLE_MESH: onoff 0x00
I (983762) BLE_MESH: event 0x00, opcode 0x8203, src 0x0001, dst 0xc000
I (983762) BLE_MESH: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (983772) BLE_MESH: onoff 0x00
W (983772) BOARD: led red is already off
```

Figura 43. Comportamiento del nodo 2

```
I (950013) BLE_MESH: event 0x00, opcode 0x8203, src 0x0001, dst 0xc000
I (950013) BLE_MESH: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (950023) BLE_MESH: onoff 0x01
I (953333) BLE_MESH: event 0x00, opcode 0x8203, src 0x0001, dst 0xc000
I (953333) BLE_MESH: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (953333) BLE_MESH: onoff 0x00
I (984973) BLE_MESH: event 0x00, opcode 0x8203, src 0x0001, dst 0xc000
I (984973) BLE_MESH: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (984973) BLE_MESH: onoff 0x00
W (984983) BOARD: led red is already off
```

Figura 44. Comportamiento del nodo 3

Observando estas figuras, es importante destacar que el origen del mensaje *Set*, el cual modifica el estado de los nodos y su *opcode* es *0x8203* porque no es respondido por un mensaje status (Tabla 3), proviene de la dirección *unicast 0x0001*, que pertenece al cliente (*smartphone*), mientras que la dirección destino es la dirección del grupo *Group – Leds Red* (*0xC000*).

Por último, en la siguiente ilustración de muestra un pequeño esquema de esta última implementación para facilitar la comprensión.

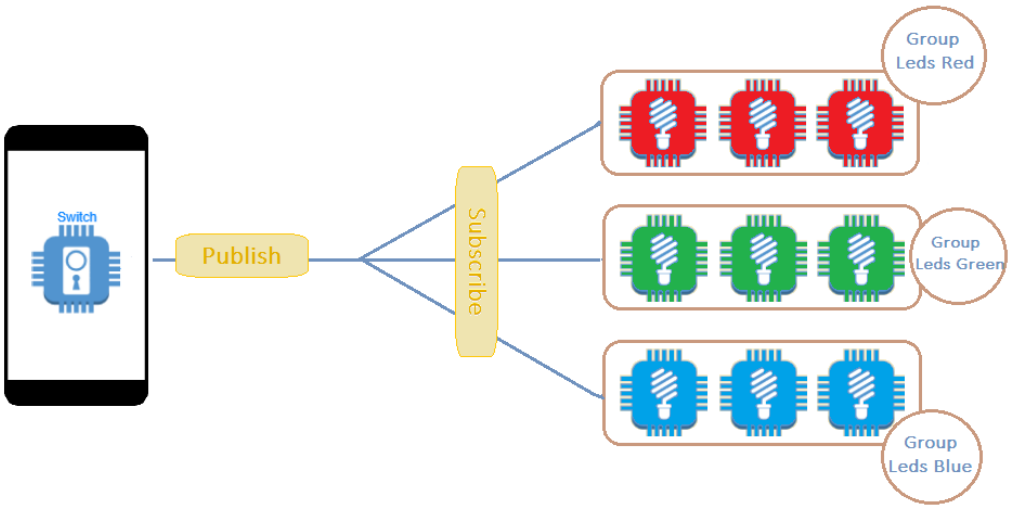

Figura 45. Esquema del control de LEDs RGB distribuidos en diferentes grupos

6.2. Desarrollo de una solución de sensorización con BLE Mesh en ESP-IDF

Los sensores desempeñan un papel fundamental en muchas aplicaciones de redes en malla, incluidas, entre otras, las que están enfocadas en edificios inteligentes. Además de medir los atributos del entorno, como pueden ser la humedad relativa o la temperatura ambiente, también pueden detectar e informar de eventos, como, por ejemplo, el estado de la ocupación de diferentes habitaciones. Por tanto, los datos recogidos por los sensores pueden usarse para influir en la toma de decisiones de una operación de un tipo particular de dispositivo, o pueden usarse para controlar o cambiar el estado de una gran cantidad de dispositivos de diferentes tipos, todo de una vez.

Como se comentaba en capítulos anteriores, los modelos son especificaciones software que, cuando se incluyen en un determinado producto, determinan qué puede hacer un dispositivo dentro de la malla. Desde el punto de vista de una red, se puede decir que los modelos hacen que un dispositivo sea lo que es.

Por tanto, en esta segunda prueba de concepto, se va a trabajar con los modelos sensores, pues están definidos por la especificación de los modelos de malla de Bluetooth SIG. Además, ESP-IDF proporciona en el directorio *esp-idf/examples/bluetooth/esp_ble_mesh/ble_mesh_sensor_model* de su GitHub [50] pequeños ejemplos de cómo empezar a trabajar con el modelo sensor cliente y servidor. Estos servirán como base para crear una solución de sensorización con Bluetooth Mesh.

6.2.1. Modelo sensor

Dentro de los modelos estándar que define Bluetooth SIG se encuentran los modelos para sensores (*Sensor Model*). En este caso, se cuenta únicamente con tres modelos, ya que los modelos *sensor client, server* y *setup*, son capaces de proporcionar un enfoque generalizado para el funcionamiento de un sensor que pertenezca a una red en malla Bluetooth. Asimismo, aprovechando las ventajas que ofrece Bluetooth Mesh, estos modelos

permiten que cualquier tipo de sensor comunique sus lecturas a otros nodos de la misma red.

El modelo *sensor server*, se utiliza para exponer una serie de estados del sensor, mientras que el modelo *setup server* es una extensión de este. Por tanto, cuando un elemento implementa el modelo *sensor server*, también tiene que tener el modelo *setup server*. Por su parte, el modelo *sensor client*, se usa para consumir los valores de estado expuestos tanto por el modelo *sensor server* como por *sensor setup server.* En la Figura 46 se ilustra la relación entre los tres modelos.

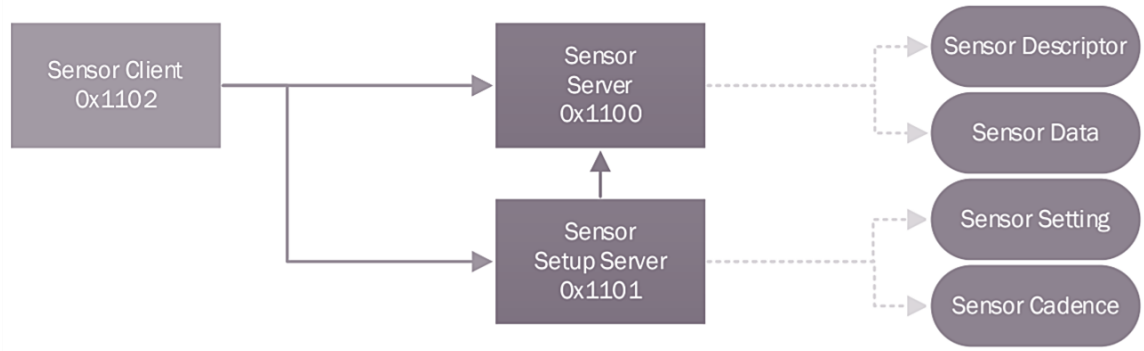

Figura 46. Relación entre los modelos sensor server, setup server y client [23]

Por otro lado, el modelo sensor hace uso de lo que se conoce como *propiedades*. Estas se diferencian de los estados en que contienen tanto un identificador como un valor, donde el identificador indica qué tipo de dato contiene dicha propiedad. El aprovechamiento de estas permite que los tres modelos se puedan acomodar a cualquier tipo de datos y sensor, en lugar de requerir diferentes mensajes y estados para cada tipo de sensor particular que forma parte de la red. Por otra parte, el modelo sensor se define alrededor de un solo estado compuesto llamado *estado del sensor*. Es un estado complejo cuyas partes principales se distribuyen entre el modelo *sensor server* y el modelo *sensor setup server* (Figura 47). A continuación, se listan los estados que componen al estado del sensor, pero antes, se puede ver el desglose completo de este en la siguiente figura.

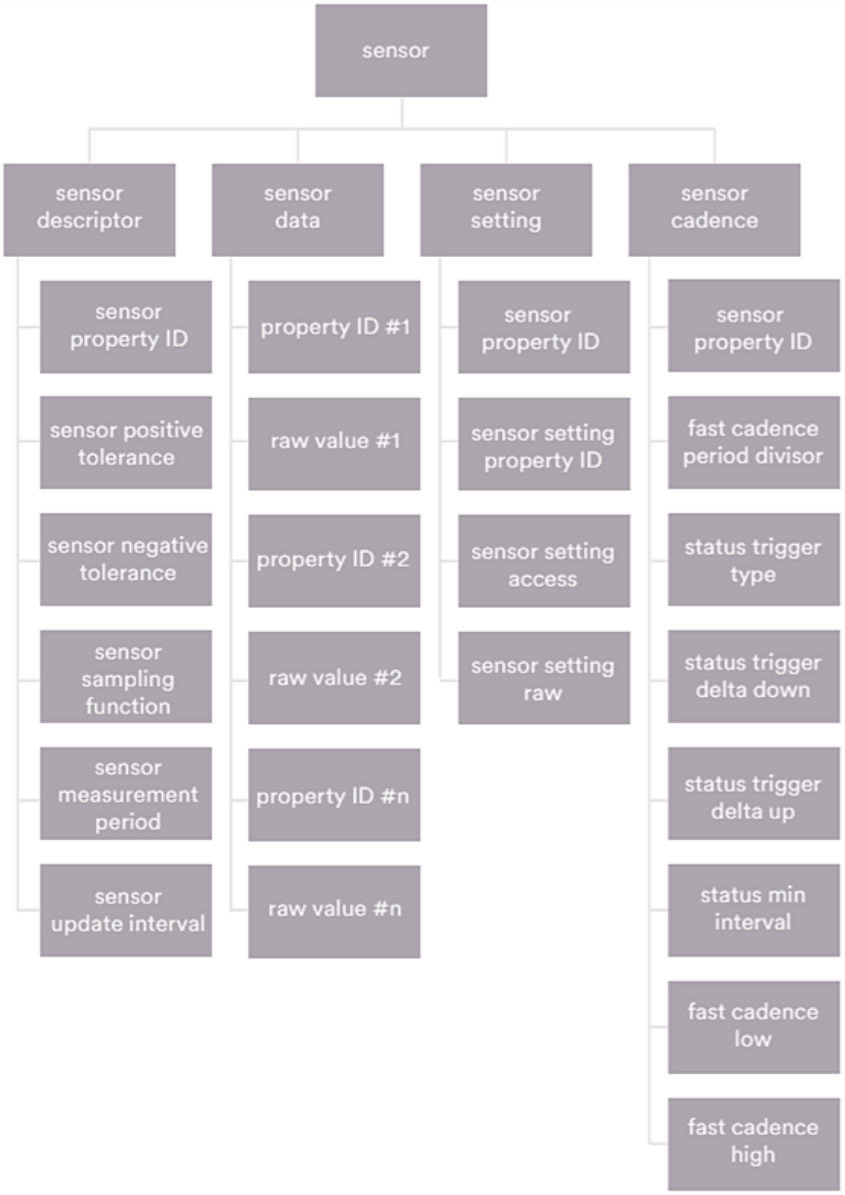

Figura 47. Estados del modelo sensor [23]

- *Sensor Descriptor state*:

  Contiene información que describe los datos del sensor disponibles. No se espera que cambie durante toda la vida útil del sensor. Los campos de tolerancia indican la magnitud de posibles errores en las mediciones tomadas por el sensor. Por su parte, el campo de la función de muestreo indica el tipo de función aplicada a los valores medidos por el sensor, por ejemplo, la media aritmética. El campo del periodo de la medición indica el periodo de tiempo durante el cual se aplica la función definida en el campo anterior. Y, finalmente, en el campo de intervalo de la actualización se indica la frecuencia con la que el sensor realiza cada medición.

- *Sensor Data state:*

  Es una secuencia de uno o más pares de campos del ID de la propiedad del sensor y un valor bruto. Las especificaciones[35] de las propiedades del dispositivo de malla Bluetooth definen las propiedades y las características con las que se relacionan. La propiedad a la que se hace referencia en un determinado dispositivo tiene un valor simple que se puede captar utilizando un mensaje del modelo sensor como *sensor get*, que devuelve el valor del estado de los datos del sensor en un mensaje *sensor status*. Por otro lado, algunas propiedades pueden definir matrices con los datos medidos por los sensores. Cuando esto ocurre se hace uso de otro estado denominado *Sensor Serie Column state*, que hace posible acceder a las columnas de estas matrices con un mensaje *sensor column get*, que devuelve un mensaje *sensor column status*.

- *Sensor Setting state:*

  Controla los parámetros de un sensor. Este estado contiene una lista de configuraciones, como, por ejemplo, los umbrales de sensibilidad, y sus valores. Cada miembro de la lista consta del ID de la propiedad a la que se aplica la configuración, el ID de una propiedad que identifica la configuración en sí, una identificación que indica si la configuración es de solo lectura o también se puede escribir en ella, y el valor de la configuración sin procesar. Por ejemplo, un sensor de ocupación puede tener una configuración de "sensibilidad" que, como su nombre indica, controla la sensibilidad del sensor. Dicho parámetro pude ajustarse para evitar que se creen falsas alarmas, accionadas, por ejemplo, por pequeños animales que activen el sensor.

- *Sensor Cadence state:*

  Permite configurar la frecuencia con la que un sensor publica informes de estado relacionados con cada tipo de datos del sensor. También permite enviar esos mensajes de estado a una frecuencia diferente para un determinado rango de valores medidos.

Por último, y a modo de curiosidad, igual que el modelo *Generic On/Off* tenía códigos de operación asociados a los distintos tipos de mensajes, los modelos *Sensor* y *Sensor Setup* también los tienen. En la Tabla 4 se muestran dichos *opcodes*, los cuales son especificados en la documentación oficial de Bluetooth Mesh [21].

[35] https://www.bluetooth.com/specifications/mesh-specifications/mesh-properties/

Server Model	Message Name	Opcode
Sensor	*Sensor Descriptor Get*	*0x82 0x30*
	Sensor Descriptor Status	*0x51*
	Sensor Get	*0x82 0x31*
	Sensor Status	*0x52*
	Sensor Column Get	*0x82 0x32*
	Sensor Column Status	*0x53*
	Sensor Series Get	*0x82 0x33*
	Sensor Series Status	*0x54*
Sensor Setup	*Sensor Cadence Get*	*0x82 0x34*
	Sensor Cadence Set	*0x55*
	Sensor Cadence Set Unacknowledged	*0x56*
	Sensor Cadence Status	*0x57*
	Sensor Settings Get	*0x82 0x35*
	Sensor Settings Status	*0x58*
	Sensor Setting Get	*0x82 0x36*
	Sensor Setting Set	*0x59*
	Sensor Setting Set Unacknowledged	*0x5A*
	Sensor Setting Status	*0x5B*

Tabla 4. Códigos de operación del modelo sensor

6.2.2. Estructura general de la solución

Para desarrollar el firmware de esta prueba de concepto se usa de nuevo el framework ESP-IDF. En esta ocasión, habrá que desarrollar dos *firmwares* diferentes, por ello, a continuación, se van a tratar dos secciones para hablar extensamente de ellos.

Para tener un esquema general de la prueba de concepto que se está desarrollando, se puede observar en la siguiente figura una ilustración de la arquitectura del sistema.

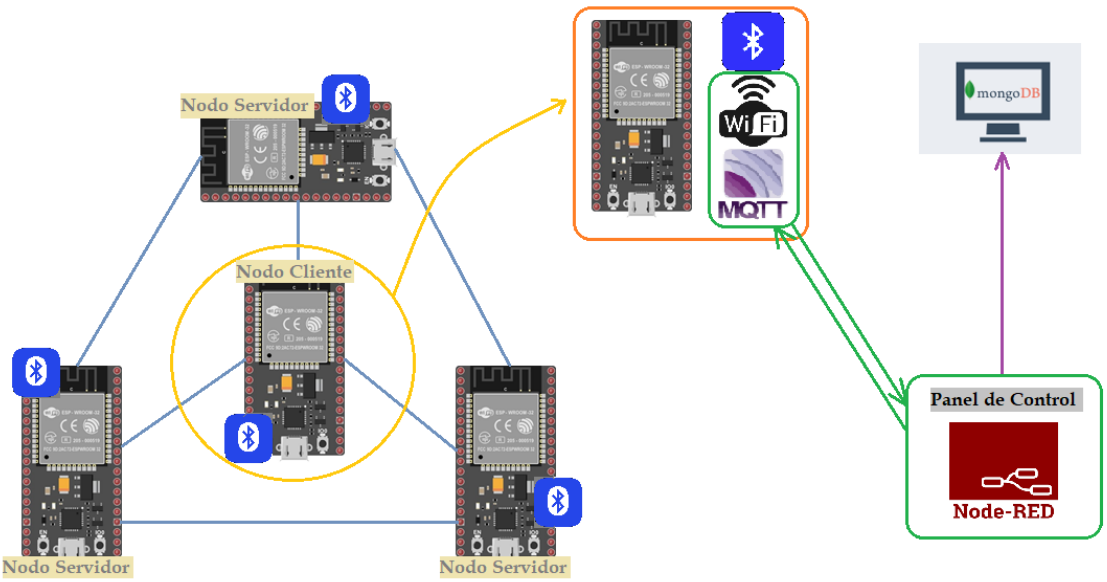

Figura 48. Esquema de la solución de sensorización con BLE Mesh

6.2.2.1. Firmware del servidor

Este firmware deberá estar flasheado en todas las placas ESP32 con las que se quiera contar en la red de malla Bluetooth. Los dispositivos estarán sin aprovisionar y contarán la implementación de un modelo *sensor server* y un modelo *sensor setup server*.

Los nodos de la malla están programados para trabajar con sensores de temperatura. En este caso, el modelo *sensor server* admite dos instancias diferentes para el estado del sensor. En primer lugar, la propiedad con ID *0x0056* representa la "Temperatura ambiente actual interior", mientras que la propiedad con ID *0x005B* representa la "Temperatura ambiente actual exterior". Ambos identificadores de la propiedad han sido tomados de la lista de especificaciones que ofrece Bluetooth SIG. Por otra parte, al no contar con sensores de temperatura reales, los datos recogidos por estos serán creados de manera sintética, es decir, a través de valores aleatorios. La temperatura interior oscilará entre 18 ºC y 30 ºC, mientras que la temperatura exterior estará entre 15 ºC y 35 ºC. En la siguiente tabla se recogen los valores de temperatura óptimos dependiendo si se han medido en el exterior o en el interior de, por ejemplo, una vivienda.

	Temperatura interior	Temperatura exterior
Temperatura insuficiente	$T^{a}{}_{medida} < 20\ ºC$	$T^{a}{}_{medida} < 18\ ºC$
Temperatura óptima	$20\ ºC \leq T^{a}{}_{medida} \leq 26\ ºC$	$18\ ºC \leq T^{a}{}_{medida} \leq 30\ ºC$
Temperatura excesiva	$T^{a}{}_{medida} > 26\ ºC$	$T^{a}{}_{medida} > 30\ ºC$

Tabla 5. Relación de temperaturas

Finalmente, hay que mencionar que el comportamiento de los servidores irá marcado por las peticiones del cliente, las cuales serán respondidas por el mensaje correspondiente. A la hora de describir el firmware del cliente esto se podrá ver en más detalle.

6.2.2.2. Firmware del cliente

Dentro de la red en malla Bluetooth, que se pretende construir, debe haber un dispositivo actuando de cliente. Su firmware contará con la implementación de un modelo *sensor client* y, además, el dispositivo actuará de aprovisionador, por tanto, no será necesario contar con un teléfono móvil como administrador de red.

Una vez que los dispositivos que están actuando como servidores sean aprovisionados y configurados con éxito, el usuario puede presionar el botón *BOOT* de la ESP32 del cliente para obtener los valores de estado correspondientes de los diferentes sensores. Dentro del fichero *board.c*, se determina en que orden se enviarán estos estados.

```
static uint32_t send_opcode[] = {
    [0] = ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_GET,
    [1] = ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_GET,
    [2] = ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_GET,
    [3] = ESP_BLE_MESH_MODEL_OP_SENSOR_GET,
    [4] = ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_GET,
};
static uint8_t press_count;
```

Figura 49. Orden de envío de los estados

Como se puede ver en la figura 49, la primera vez que se pulse el botón se enviarán cada tres segundos el estado *sensor descriptor* de los diferentes nodos sensores. La siguiente vez que se pulse se enviarán los estados *sensor cadence* y después *sensor settings*. La cuarta vez que se pulse el botón se obtendrá el estado *sensor data*, que es el que verdaderamente interesa en esta prueba. Cada uno de los servidores enviará de manera aleatoria tanto una temperatura interior como una exterior. Y, finalmente, si se pulsa una quinta vez el botón *BOOT* se enviará el estado *sensor series column*, que no es más que los valores medidos por los sensores organizados en forma de matriz. En caso de que se volviese a pulsar el botón del dispositivo del cliente, el ciclo volvería a repetirse.

Con esta prueba de concepto se pretende implementar el *modelo sensor* en BLE Mesh desde principio a fin, incluyendo otros protocolos que coexistan con Bluetooth Mesh en el mismo dispositivo y así conseguir que los datos del cliente sean extraídos hacia un servidor externo.

El repositorio de ESP-IDF contiene una gama muy amplia de proyectos desarrollados en este framework. Estos están destinados a demostrar las diferentes funcionalidades de ESP-IDF y de proporcionar un código base para poder adaptarlo en nuestros propios proyectos. Tras estudiar los proyectos de ejemplo en los que se ilustra el funcionamiento de Wi-Fi y de MQTT, se ha incluido en el proyecto del *modelo sensor* ambas tecnologías para que la ESP32, que actúa como cliente, funcione a la vez como aprovisionador de nuevos nodos, como cliente del *modelo sensor* consultado datos de sensorización de los nodos pertenecientes a la red en malla y como cliente MQTT, el cual extrae datos hacia un servidor externo gracias a la conexión Wi-Fi.

Hacer que las tres tecnologías coexistan en un mismo dispositivo se consigue a través de hilos, ya que estos permiten que se puedan realizar varias tareas a la vez.

```
/* Initialize the Bluetooth Mesh Subsystem */
nodes = 0;

err = ble_mesh_init();
if (err != ESP_OK) {
    ESP_LOGE(TAG, "Bluetooth mesh init failed (err %d)", err);
}

/* Wifi */
ESP_LOGI(TAG_wifi, "ESP_WIFI_MODE_STA");
pthread_t t1;
pthread_create(&t1, NULL, (void*) wifi_init_sta, NULL);

/* MQTT */
ESP_LOGI(TAG_mqtt, "MQTT_CLIENT");
pthread_t t2;
pthread_create(&t2, NULL, (void*) mqtt_app_start, NULL);
```

Figura 50. Inicialización de la coexistencia de las tres tecnologías

Como se ha visto en este año de máster, el Internet de las Cosas necesita protocolos estándar, sin embargo, el tiempo pasa y no acaba de alzarse ninguno sobre los demás. Al estar en esta situación, depende del desarrollador elegir el protocolo que más se ajuste a las necesidades de su proyecto. En este caso, como se ha mencionado antes, se va a trabajar con MQTT.

MQTT (*Message Quere Telemetry Transport*), es un protocolo diseñado por IBM, basado en TCP/IP y pensado para la comunicación *Machine-to-Machine* (M2M). Se centra en el envío de datos donde se requiere poco ancho de banda, por lo que es un protocolo idóneo para la comunicación de sensores dentro del ecosistema IoT.

Este protocolo sigue un modelo de publicación/suscripción al igual que BLE Mesh, solo que MQTT tiene una arquitectura en forma de estrella. Al centro de esta topología en estrella se le llama *broker*, y es el encargado de gestionar la red, es decir, a él le llegan los mensajes publicados por ciertos nodos y también renvía dichos mensajes a los subscriptores correspondientes.

En este proyecto, se ha usado el *broker* de código abierto *Eclipse Mosquitto*[36], ya que es ampliamente utilizado por su ligereza. En concreto, se ha utilizado *test.mosquitto.org*, un servidor que se proporciona para que la comunidad realice pruebas y que es totalmente suficiente para la prueba de concepto que se pretenden realizar.

Por otro lado, dentro de la arquitectura MQTT hay que incluir el concepto *topic*, pues es una cadena de caracteres que funciona como canal virtual que conecta emisores y receptores, puesto que ambos deben estar suscritos a un mismo *topic* para que se realice con éxito la comunicación. Los *topics* se estructuran de manera jerárquica, por ejemplo, un par de *topics* que se usan en la implementación de esta solución son: */temperatura/interior/nodoX* y *temperatura/exterior/nodoX*, siendo *X* el número de un determinado nodo aprovisionado. Si un usuario quisiese suscribirse a varios *topics* se podría usar el *multi-level wildcard* /#/ o el *single level wildcard* /+/. Si el usuario se suscribiese a */temperatura/interna/#*, recibiría mensajes de la temperatura interna de todos los nodos

[36] https://mosquitto.org/

pertenecientes a la red, mientras que si el usuario se suscribiese a */temperatura/+/nodo1*, recibiría tanto en valor de la temperatura interna y externa del sensor número 1.

Además, existen tres niveles de calidad de servicio (*QoS*) o tres modos de funcionamiento en MQTT:

- *At most once (QoS = 0)*: es el modo más rápido, pero menos fiable, ya que no habría garantía de entrega.

- *Al least once (QoS = 1)*: en este modo habría garantía de que el mensaje ha sido entregado al menos una vez, pero es posible que esté repetido.

- *Exactly once (QoS = 2)*: es el modo seleccionado para el proyecto puesto que es el más fiable. Controla los duplicados para garantizar que el mensaje es entregado una sola vez.

Finalmente, es importante destacar que al integrar estas tres tecnologías en un mismo nodo se obtienen una serie de errores que indican que la aplicación que se ha creado excede el tamaño máximo de los tamaños de partición ESP-IDF predeterminados. Por tanto, es necesario editar la tabla de particiones de la ESP32 creando un archivo *csv* donde se especifiquen los nuevos tamaños de la partición. Siguiendo con las recomendaciones aplicadas en otros proyectos donde coexisten varias tecnologías, se decide crear el siguiente archivo denominado como *partitios.csv*.

```
# Name,         Type,   SubType,    Offset,     Size,   Flags
# Note: if you have increased the bootloader size, make sure to update the offsets to avoid overlap
nvs,            data,   nvs,        0x9000,     16k
otadata,        data,   ota,        0xd000,     8k
phy_init,       data,   phy,        0xf000,     4k
factory,        app,    factory,    0x10000,    2M
```

Figura 51. Tabla de particiones de la ESP32 (nodo cliente)

6.2.3. Desarrollo de un panel de control vía Node-RED

Como bien se comentaba en la sección anterior, se busca obtener un despliegue BLE Mesh implementado de principio a fin del modelo sensor. A través de MQTT se van a extraer los datos del dispositivo del cliente hacia un panel de control creado en Node-RED. Antes de proceder al desarrollo del panel, se va a introducir brevemente Node-RED para asentar las bases del contexto en el que se va a trabajar.

Node-RED es una herramienta de código abierto que inicialmente fue desarrollada por IBM. Está orientada a flujos de datos, y, a través de una serie de mecanismos, permite interconectar dispositivos hardware, APIs y servicios online dentro de un mismo entorno IoT. Esta herramienta gráfica, utilizable desde cualquier navegador web, permite la creación y edición de flujos de datos que tomen datos de entrada (mediante nodos de entrada), los procesen (mediante nodos de procesamiento) y proporcionen salidas (mediante nodos de salida). Además, Node-RED permite la interconexión de elementos software y hardware mediante cualquier protocolo conocido, en este caso, MQTT, facilitando así el despliegue de infraestructuras IoT.

Las aplicaciones en Node-RED se modelan como flujos entre componentes. A continuación, se van a comentar los diferentes flujos creados, los cuales darán pie a un panel de control.

El primer flujo se muestra en la Figura 52. Inicialmente, cuenta con un nodo MQTT de entrada, que está suscrito al *topic /aprovisionado/nodo* y que obtiene datos relevantes que publica la ESP32, que actúa como cliente dentro de la red BLE Mesh, en ese mismo *topic*. La información que se comparte en el *payload* del mensaje es el número de nodo que se le ha asignado al dispositivo a la hora del aprovisionamiento y la dirección MAC del dispositivo en cuestión. Por otro lado, gracias al nodo *split* se consigue dividir el contenido del mensaje en número de nodo y dirección del dispositivo para después poder mostrarlos de manera independiente en el panel de control.

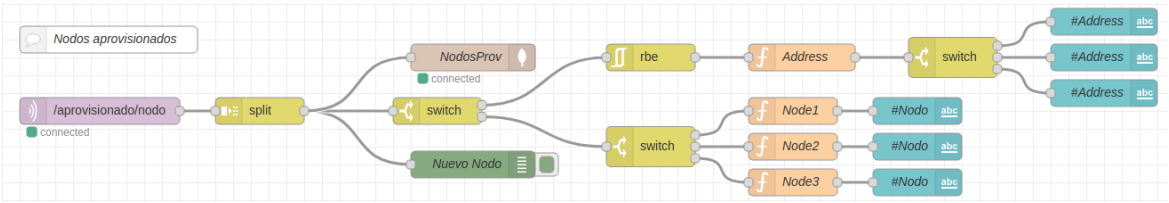

Figura 52. Flujo de recepción de nodos aprovisionados

Por otro lado, en la Figura 53, se muestran los dos flujos desarrollados que permiten la recepción a través de MQTT de la temperatura interior y la temperatura exterior medida por todos los nodos pertenecientes a la red. Esto se consigue gracias a la suscripción a los *topics /temperatura/interior/#* y */temperatura/exterior/#*. Como se verá más adelante, todos estos datos obtenidos son representados de manera muy visual en el panel de control.

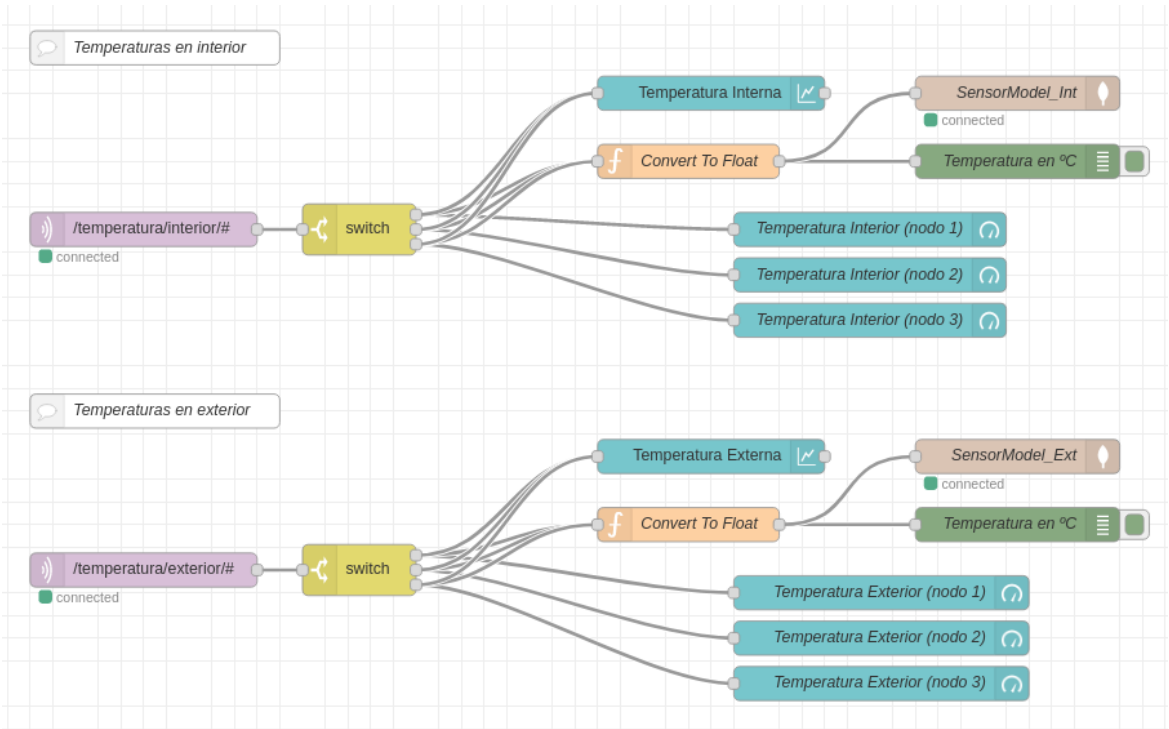

Figura 53. Flujos de recepción de temperaturas

Por otra parte, si se observan las figuras 52 y 53, se tiene que los diferentes flujos cuentan con un nodo de la base de datos *MongoDB*. Por un lado, se almacenarán en dicha base de datos los mensajes recibidos por MQTT a través del *topic /aprovisionado/nodo*, es decir, se guardarán en *MongoDB* tanto el número de nodo como la dirección del dispositivo de todos los nodos que se vayan aprovisionado. También se almacenará la temperatura interior y exterior en la base de datos, pero siguiendo el formato que marca el nodo denominado *Convert To Float*. Como se puede ver a continuación, además de almacenarse el valor medido por el sensor, se guarda la hora en la que se tomó la medida.

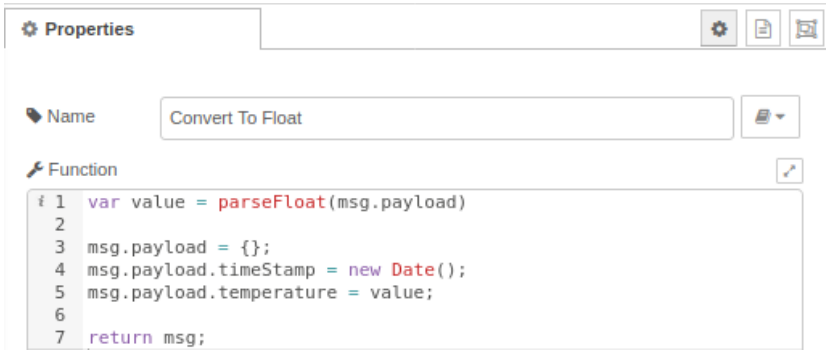

Figura 54. Conversión del formato

MongoDB[37] es una base de datos NoSQL, basada en documentos y de código abierto. Al haber trabajado con ella en otras asignaturas del máster, es interesante incorporarla en la solución que se está desarrollando de sensorización. Por su parte, *Clever Cloud* [38] es un proveedor de soluciones PaaS y permite crear de manera gratuita un clúster *MongoDB* para uso de prueba y desarrollo.

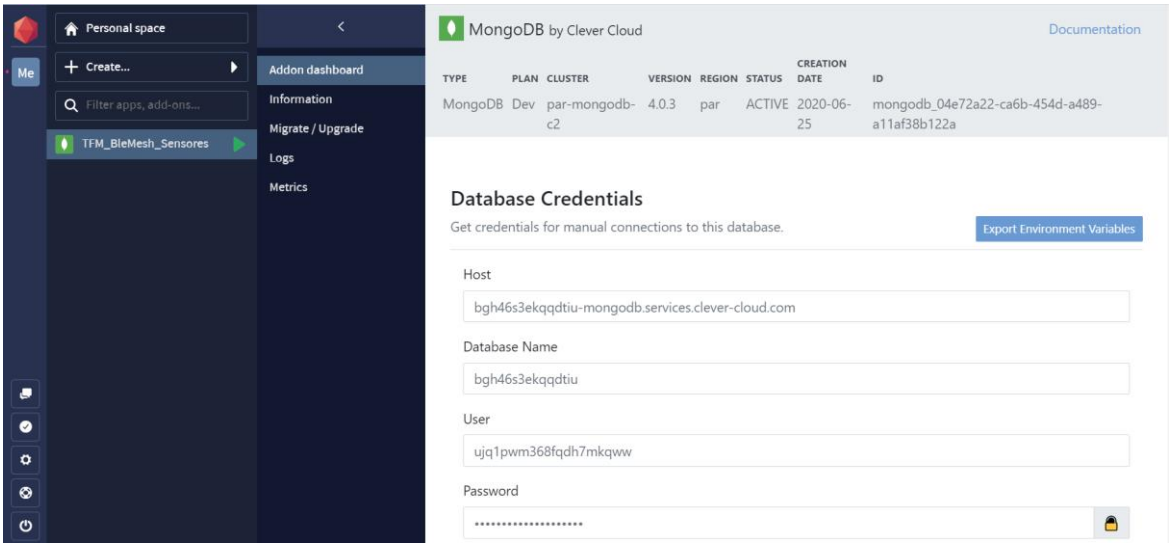

Figura 55. Creación de una base de datos MongoDB en Clever Cloud

Proporcionando en Node-RED a los nodos de salida de *MongoDB* la información correspondiente de nombre del *host*, nombre de la base datos, el usuario y la contraseña, se consigue con éxito que la información con valor en esta prueba de concepto quede almacenada en la nube.

[37] https://www.mongodb.com/
[38] https://www.clever-cloud.com/en/

Por otro lado, *MongoDB Compass*[39] es una herramienta que se instala de manera local en nuestro equipo y que muestra información sobre bases de datos *MongoDB* para poder realizar consultas sobre ellas de manera muy sencilla y visual. Esta herramienta permite conectarse a una base de datos *MongoDB* existente, en este caso, la que se creó en la nube gracias a *Clever Cloud*. Dentro de la propia base de datos se irán creando colecciones, dependiendo de los datos que se vayan a almacenar. Esto se podrá ver mejor en la siguiente sección.

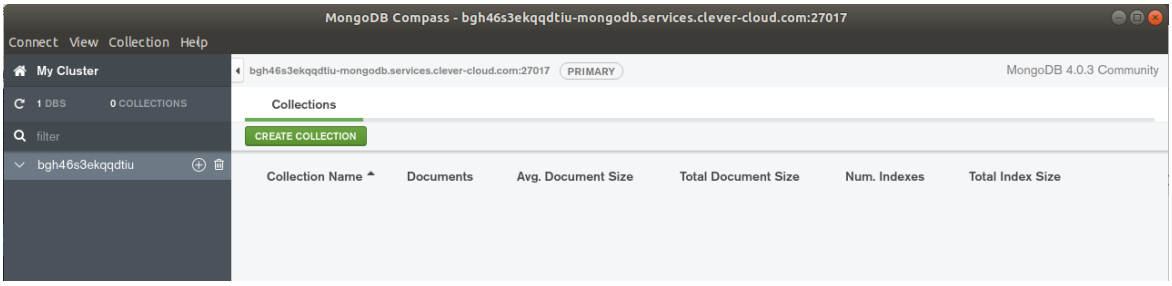

Figura 56. Visualización de la base de datos desde MongoDB Compass

Volviendo a los flujos de Node-RED, se han creado dos pequeños flujos (Figura 57) que hacen que el panel de control no reporte únicamente de manera pasiva. Hasta ahora, en el panel de control se muestran los nodos aprovisionados con sus correspondientes direcciones del dispositivo y las diferentes temperaturas obtenidas cuando el usuario presiona (por cuarta vez) el botón *BOOT* de la ESP32 que actúa como cliente de la red BLE Mesh. En este caso, desde el panel de control, el usuario podrá solicitar la temperatura interior o exterior del nodo que desee. De este modo, cada vez que se pulse el botón asociado a un sensor en el panel de control, se envía un mensaje MQTT al cliente bajo el *topic* */solicitud/tempInt* o */solicitud/tempExt*, solicitando así el nuevo dato al servidor correspondiente y devolviendo este por los *topics* */temperatura/interior/nodoX* y */temperatura/exterior/nodoX*, siendo *nodoX* el número de nodo en concreto del que se desea actualizar el valor. Con esto simplemente lo que se consigue en no tener que pulsar el botón desde el aprovisionador para solicitar los datos de todos los nodos servidores que forman parte de la red, sino habilitar la posibilidad de poder hacerlo desde Node-RED.

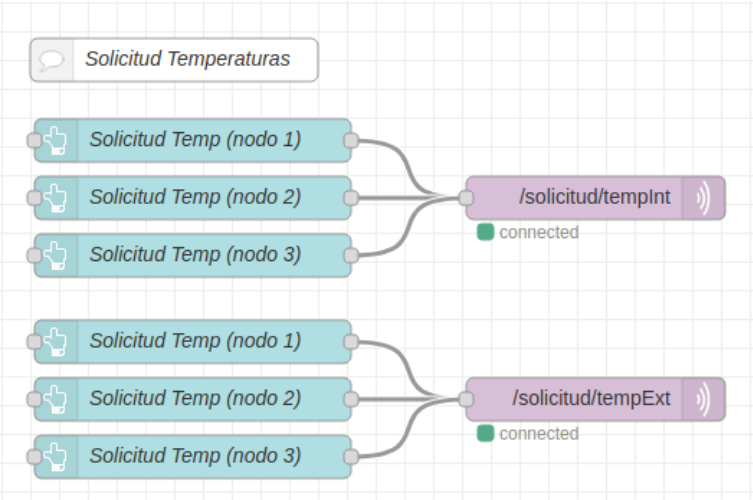

Figura 57. Flujo de solicitud de temperaturas por parte del usuario

[39] https://www.mongodb.com/products/compass

Finalmente, la Figura 58 ilustra cómo se vería el panel de control. A pesar de estar vacío, se puede observar como este consta de dos pestañas. La pestaña principal es *Sensores Aprovisionados*, en la que aparecerán los nodos en el orden que han entrado a formar parte de la red junto con la dirección MAC del dispositivo. Además, se podrá visualizar las diferentes temperaturas y desde aquí el usuario podrá solicitar información actualizada de estas a través de los diferentes botones de los distintos sensores. Por su parte, en la pestaña *Monitorización de la Temperatura*, se podrá visualizar gráficamente cómo ha ido variando la temperatura a lo largo de las diversas pruebas en el tiempo.

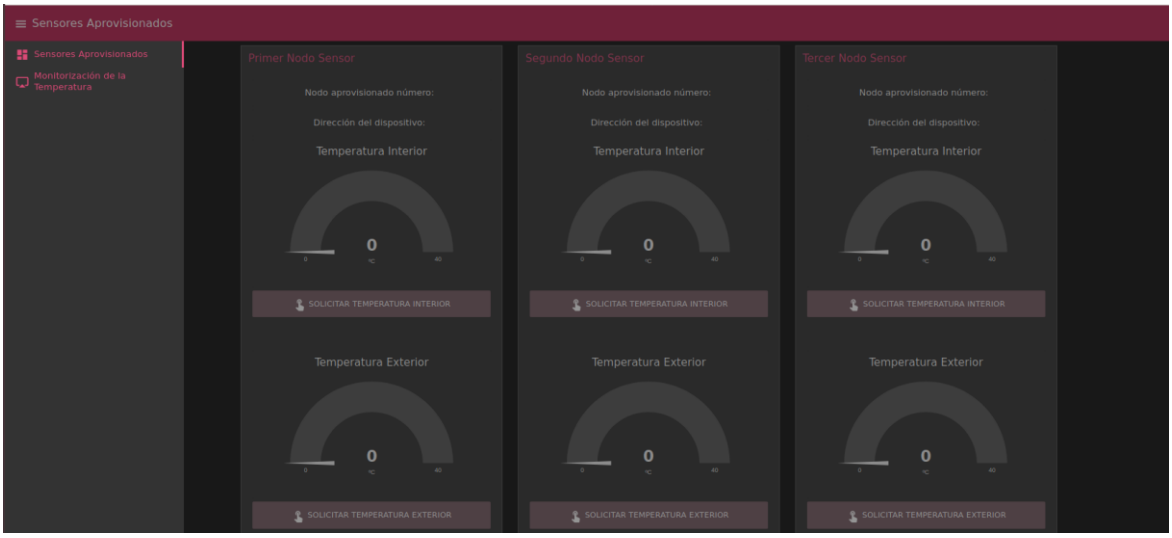

Figura 58. Panel de Control desarrollado con Node-RED

6.2.4. Pruebas de uso

Llegado este punto, se podría afirmar que se ha conseguido desarrollar un despliegue BLE Mesh del modelo sensor, el cual incluye tanto un panel de control como una base de datos en la que se van almacenando los valores recogidos por los diferentes sensores. Ahora, por tanto, es el turno de realizar en este cuarto punto las pruebas necesarias para demostrar el correcto funcionamiento de este.

En primer lugar, el firmware desarrollado para el servidor debe estar flasheado en todas las ESP32 que vayan a formar parte de la futura red Bluetooth Mesh adquiriendo el rol de sensores de temperatura. Por su parte, antes de flashear el firmware del cliente en la ESP32 seleccionada para que, entre otras cosas, actúe como aprovisionador, hay que configurar a través del comando *idf.py menuconfig* la dirección del *broker* MQTT y la red Wi-Fi a la que la placa debe estar conectada para que esta tecnología de comunicación funcione. Además, desde ese mismo menú habrá que indicarle al firmware el archivo que especifica los nuevos tamaños de la partición que debe adquirir la ESP32. En las siguientes imágenes se muestra una ruta a seguir por este menú de configuración.

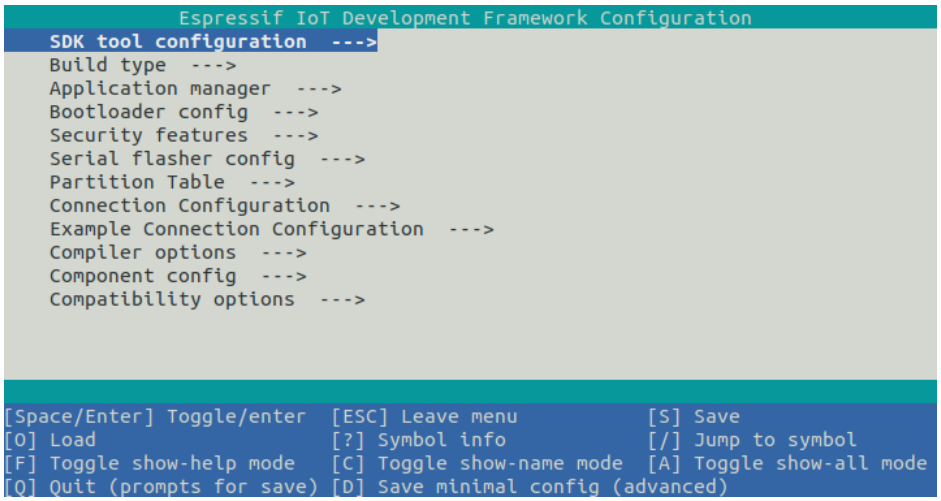

Figura 59. Pantalla de inicio del menú de configuración

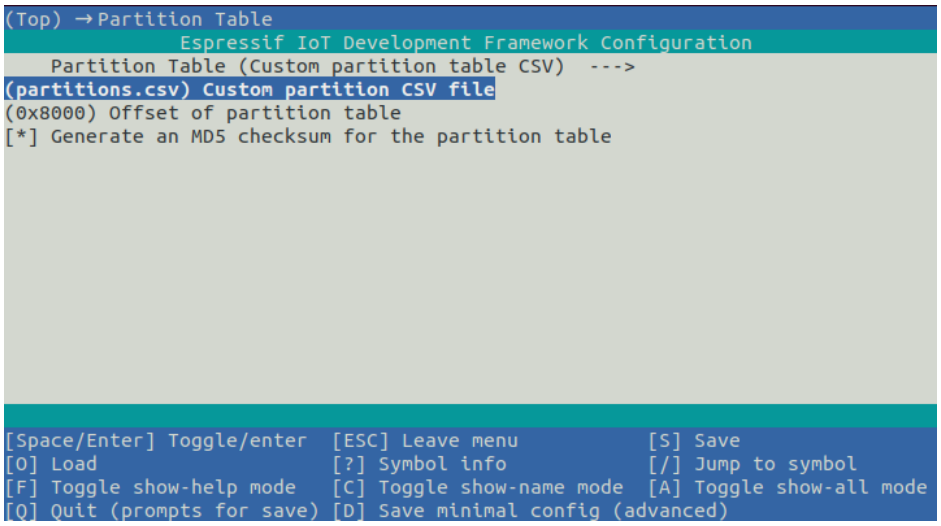

Figura 60. Cambio de los tamaños de la partición de la placa

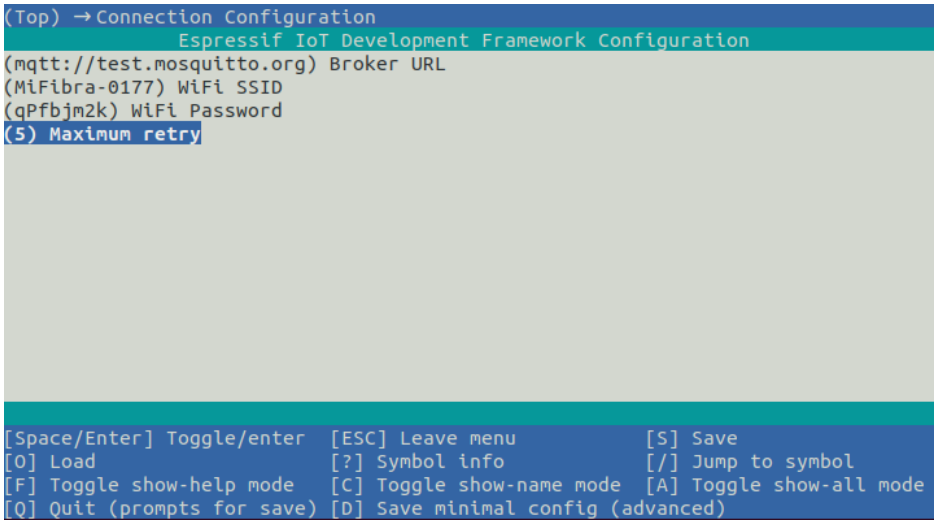

Figura 61. Configuración del broker MQTT y de la red Wi-Fi

A continuación, al monitorear lo que ocurre en el cliente con el comando *idf.py -p /dev/ttyUSB0 monitor*, se puede ver que este ya está listo para empezar el aprovisionamiento de nodos a la red, pues lo indica con el mensaje *BLE Mesh sensor client initialized*.

Cuando los dispositivos servidores se encienden, el cliente automáticamente detecta, pues los servidores mandan mensajes de *advertising* para publicitarse, que hay posibles nodos a su alcance para proceder a su aprovisionamiento. Por ejemplo, en la siguiente figura se muestra en el monitor del cliente cómo este aprovisiona y configura satisfactoriamente el primer nodo de la red. Para ello, recoge la dirección del dispositivo, le asigna una dirección para su primer elemento, siendo esta la *0x0005*, y finalmente le asigna el nombre de *nodo 0*.

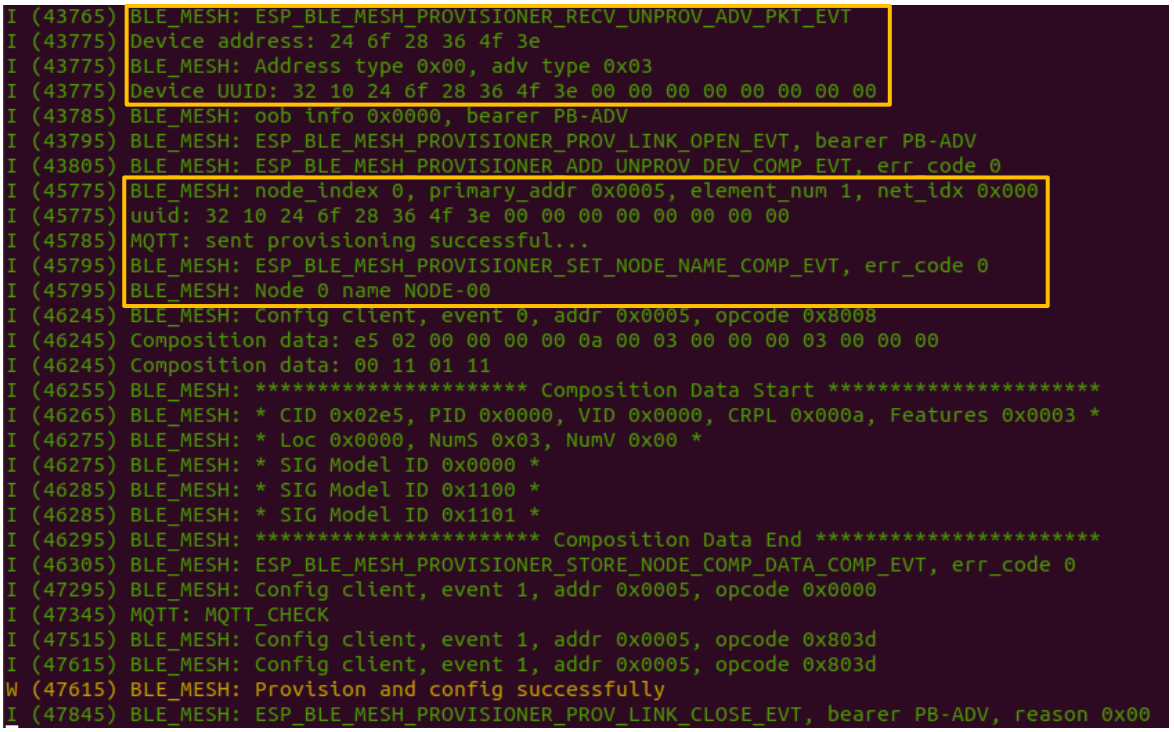

Figura 62. Aprovisionamiento del primer nodo a la red BLE Mesh (nodo cliente)

Al mismo tiempo, al monitorear lo que ocurre en este primer nodo aprovisionado, se ve como se le informa de que la dirección que le ha sido asignada fue la *0x0005* y además se le ha sido enviada la *AppKey*.

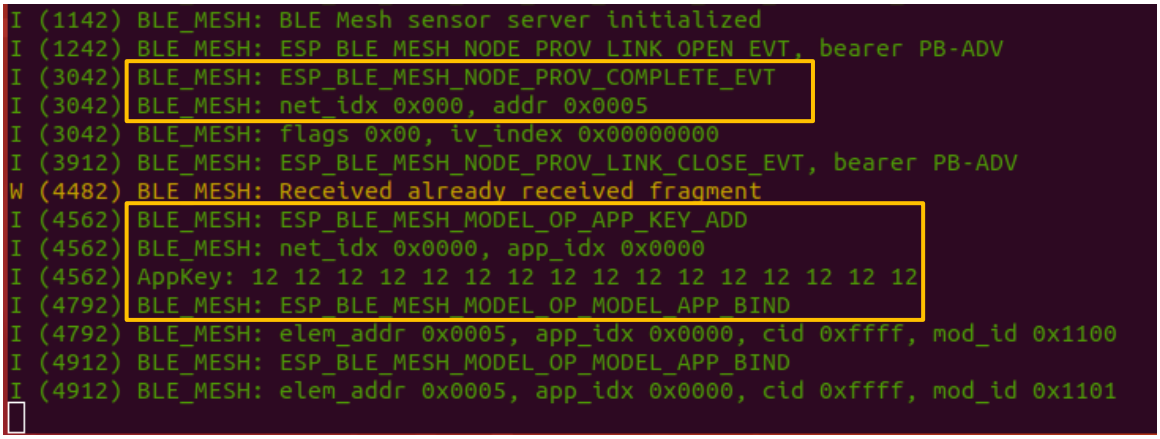

Figura 63. Aprovisionamiento del primer nodo a la red BLE Mesh (nodo servidor)

Después, ocurre lo mismo con las otras dos placas, cuyas direcciones *unicast* fueron *0x0006* y 0x*0007*.

Al presionar el botón *BOOT* de la placa con el rol de cliente se obtienen los diferentes estados que se comentaros anteriormente de nos nodos servidores de la red. En primer lugar, llega *sensor descriptor status*, después, *sensor cadence status* y *sensor settings status*. Tras estos se recibe *sensor data status* y, finalmente *sensor series columna status*.

```
I (48675) BLE_MESH: Waiting a bit...
I (48825) BLE_MESH: Sensor client, event 0, addr 0x0005
I (48825) BLE_MESH: Sensor Descriptor Status, opcode 0x0051
I (48825) Sensor Descriptor: 56 00 00 00 00 00 00 00 5b 00 00 00 00 00 00 00
I (51675) BLE_MESH: Done...
I (51675) BLE_MESH: Waiting a bit...
I (51865) BLE_MESH: Sensor client, event 0, addr 0x0006
I (51865) BLE_MESH: Sensor Descriptor Status, opcode 0x0051
I (51865) Sensor Descriptor: 56 00 00 00 00 00 00 00 5b 00 00 00 00 00 00 00
I (54675) BLE_MESH: Done...
I (54675) BLE_MESH: Waiting a bit...
I (54845) BLE_MESH: Sensor client, event 0, addr 0x0007
I (54845) BLE_MESH: Sensor Descriptor Status, opcode 0x0051
I (54845) Sensor Descriptor: 56 00 00 00 00 00 00 00 5b 00 00 00 00 00 00 00
I (57675) BLE_MESH: Done...
```

Figura 64. Obtención del estado sensor descriptor de los nodos de la red BLE Mesh (nodo cliente)

```
I (304085) BLE_MESH: Waiting a bit...
I (304125) BLE_MESH: Sensor client, event 0, addr 0x0005
I (304135) BLE_MESH: Sensor Cadence Status, opcode 0x0057, Sensor Property ID 0x0056
I (307085) BLE_MESH: Done...
I (307085) BLE_MESH: Waiting a bit...
I (307135) BLE_MESH: Sensor client, event 0, addr 0x0006
I (307135) BLE_MESH: Sensor Cadence Status, opcode 0x0057, Sensor Property ID 0x0056
I (310085) BLE_MESH: Done...
I (310085) BLE_MESH: Waiting a bit...
I (310115) BLE_MESH: Sensor client, event 0, addr 0x0007
I (310115) BLE_MESH: Sensor Cadence Status, opcode 0x0057, Sensor Property ID 0x0056
I (313085) BLE_MESH: Done...
```

Figura 65. Obtención del estado sensor cadence de los nodos de la red BLE Mesh (nodo cliente)

```
I (334985) BLE_MESH: Waiting a bit...
I (335075) BLE_MESH: Sensor client, event 0, addr 0x0005
I (335075) BLE_MESH: Sensor Settings Status, opcode 0x0058, Sensor Property ID 0x0056
I (337985) BLE_MESH: Done...
I (337985) BLE_MESH: Waiting a bit...
I (338035) BLE_MESH: Sensor client, event 0, addr 0x0006
I (338035) BLE_MESH: Sensor Settings Status, opcode 0x0058, Sensor Property ID 0x0056
I (340985) BLE_MESH: Done...
I (340985) BLE_MESH: Waiting a bit...
I (341095) BLE_MESH: Sensor client, event 0, addr 0x0007
I (341095) BLE_MESH: Sensor Settings Status, opcode 0x0058, Sensor Property ID 0x0056
I (343985) BLE_MESH: Done...
```

Figura 66. Obtención del estado sensor settings de los nodos de la red BLE Mesh (nodo cliente)

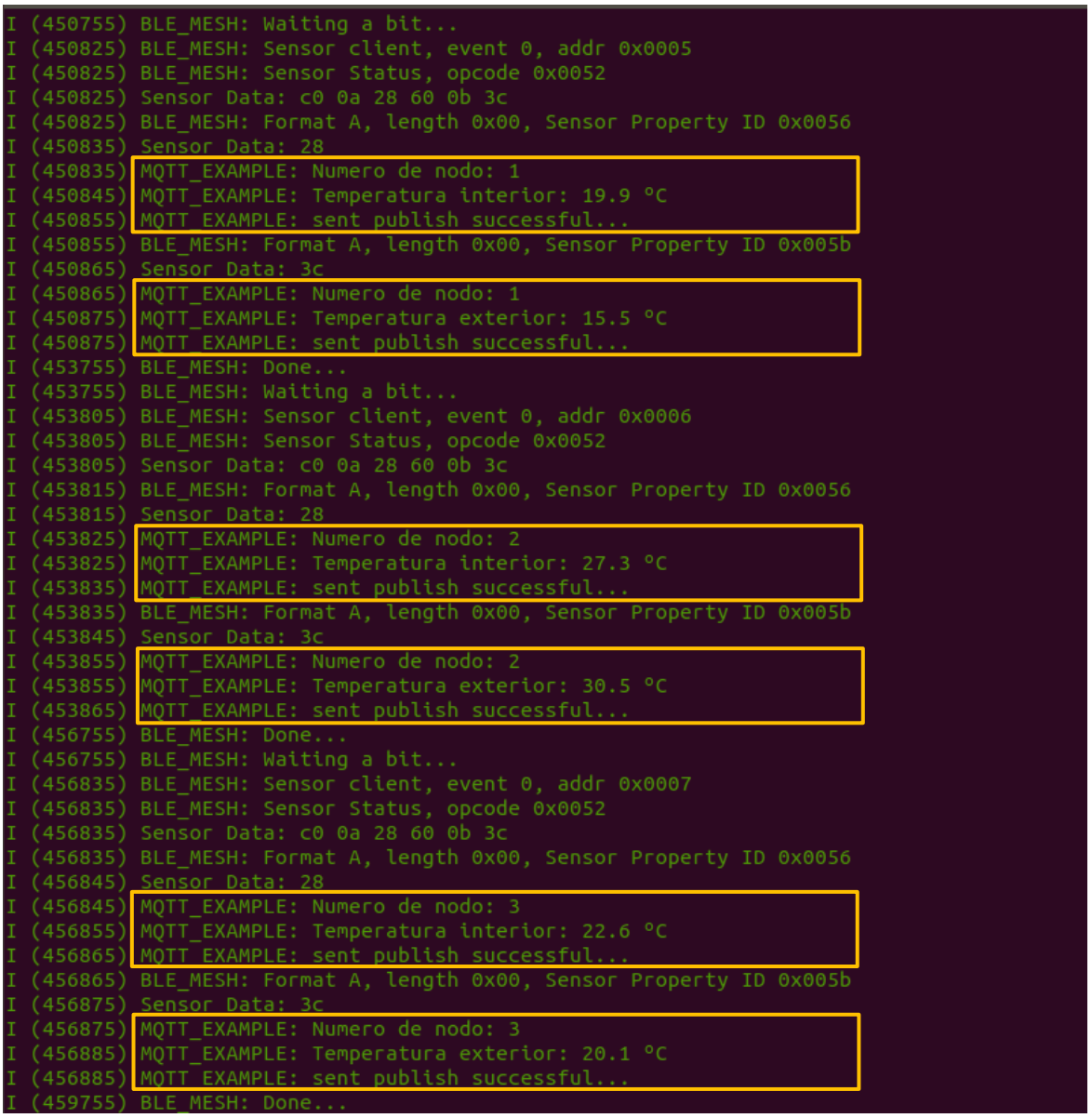

Figura 67. Obtención del estado sensor data de los nodos de la red BLE Mesh (nodo cliente)

```
I (555355) BLE_MESH: Waiting a bit...
I (555455) BLE_MESH: Sensor client, event 0, addr 0x0005
I (555455) BLE_MESH: Sensor Series Status, opcode 0x0054, Sensor Property ID 0x0056
I (558355) BLE_MESH: Done...
I (558355) BLE_MESH: Waiting a bit...
I (558395) BLE_MESH: Sensor client, event 0, addr 0x0006
I (558395) BLE_MESH: Sensor Series Status, opcode 0x0054, Sensor Property ID 0x0056
I (561355) BLE_MESH: Done...
I (561355) BLE_MESH: Waiting a bit...
I (561445) BLE_MESH: Sensor client, event 0, addr 0x0007
I (561445) BLE_MESH: Sensor Series Status, opcode 0x0054, Sensor Property ID 0x0056
I (564355) BLE_MESH: Done...
```

Figura 68. Obtención del estado sensor series de los nodos de la red BLE Mesh (nodo cliente)

En realidad, con el estado que se pretende trabajar es con *sensor data* (Figura 67), ya que el contenido de los otros estados, en esta prueba, es enviado por defecto, aunque si se trabajase con sensores reales y en un proyecto real, estos podrían configurarse.

Como puede verse en la Figura 67, cada servidor envía por Bluetooth el dato correspondiente a la propiedad indicada. Por ejemplo, para el primer nodo de la red, se indica el ID de la propiedad, es decir, *0x0056*, y a continuación el cliente informa de que se trata del nodo 1 y que el valor recogido por el sensor de temperatura interior es de 19.9 ºC. Ocurre lo mismo para la propiedad *0x005b* del nodo 1. Después, todos estos datos son enviados por MQTT al panal de control desarrollado en Node-RED.

Como se puede ver a continuación, el panel de control se compone de 3 paneles independientes, pues cada uno pertenece a un nodo. En la parte superior de estos se indica el número asignado al nodo y la dirección MAC del propio dispositivo. De esta manera se comprueba que ninguno ha sido duplicado. Después, se representa la temperatura interna del nodo en cuestión y la temperatura externa, habiendo debajo de estas un botón del que se hablará más adelante.

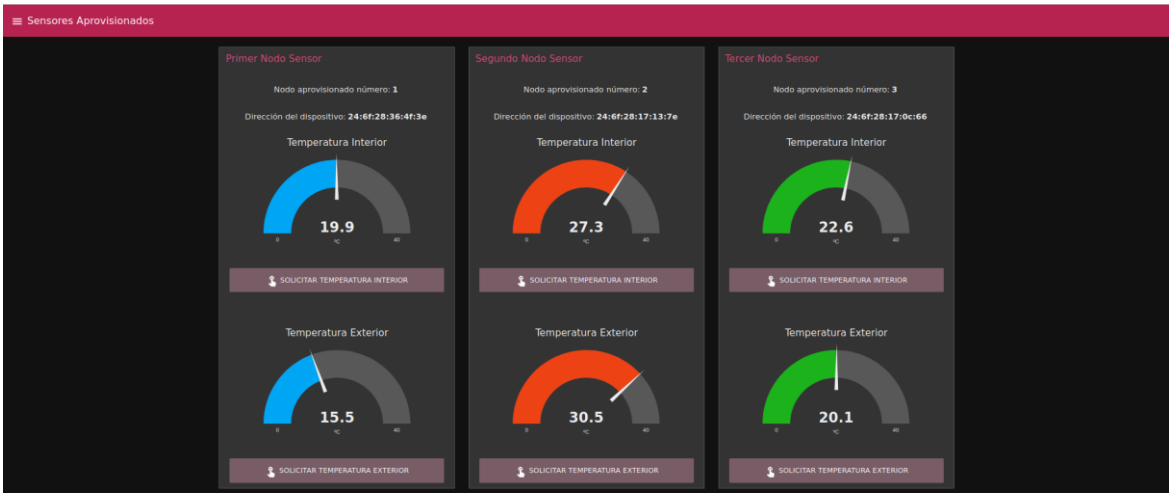

Figura 69. Recepción de las temperaturas en el panel de control

Recordando los umbrales determinados en la Tabla 5, se tiene que el panel de control representa de verde la temperatura óptima, en este caso, las del nodo 3, puesto que cuenta con ambas temperaturas dentro de su umbral óptimo. En azul, se representa las temperaturas con valores inferiores a lo ideal, en este caso, es lo que sucede con el nodo 1, y, finalmente, cuando las temperaturas son superiores a lo establecido como "correcto", estas se representan en un tono rojizo, como ocurre con el nodo 2.

Por otro lado, al iniciar *MongoDB Compass,* se puede comprobar que la base de datos que se tiene de *MongoDB* en la nube de *Clever Cloud* contiene dos nuevas colecciones.

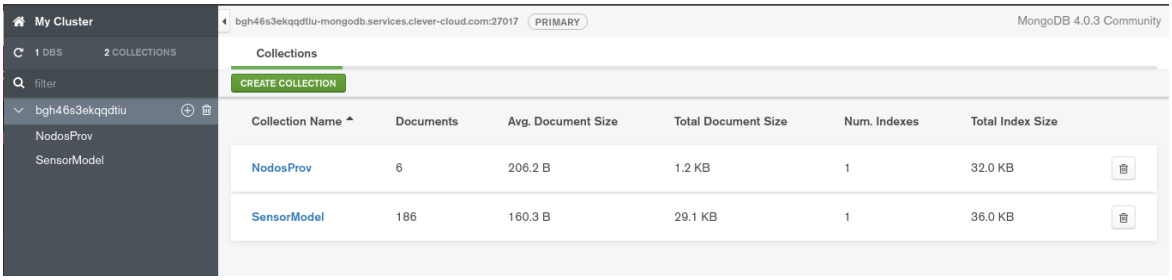

Figura 70. Colecciones en MongoDB Compass

La primera de ellas, denominada como *NodosProv*, cuenta con 6 elementos. Como se puede ver en la Figura 71, tres de estos indican el número de nodo aprovisionado y los otros tres indican la dirección MAC de los dispositivos servidores de la red en malla. En esta prueba, esta colección no aumentará su tamaño puesto que no se cuenta con más placas, sin embargo, la otra colección sí lo hará.

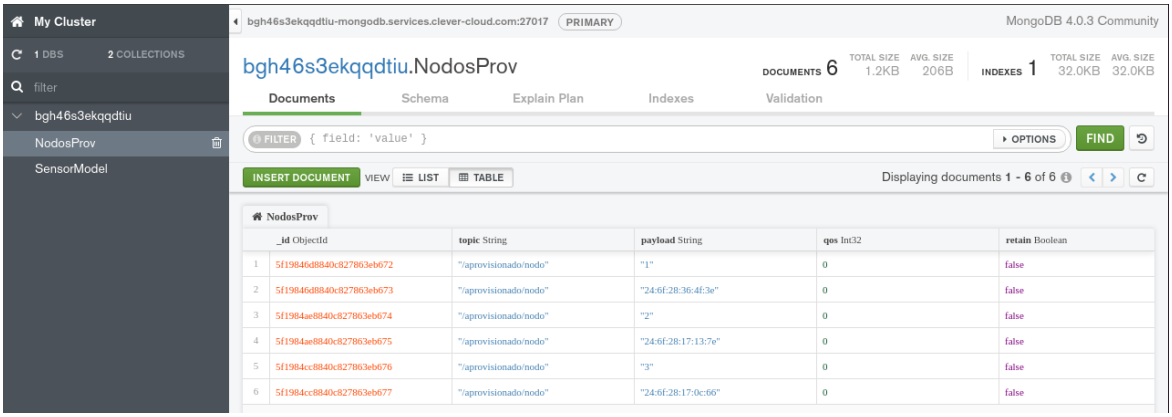

Figura 71. Colección NodosProv en MongoDB Compass

La colección *SensorModel* hasta ahora cuenta con 186 documentos. Estos pueden ir aumentando siempre que el usuario decida solicitar al cliente las temperaturas adquiridas por los diferentes nodos servidores de la red. Como se ve en la Figura 72, cada documento cuenta con un *id* único, contiene el *topic* desde el que se obtuvo el valor y, además, al desplegar el *payload* se muestra la fecha y hora a la que se adquirió el dato y el dato de la temperatura en sí.

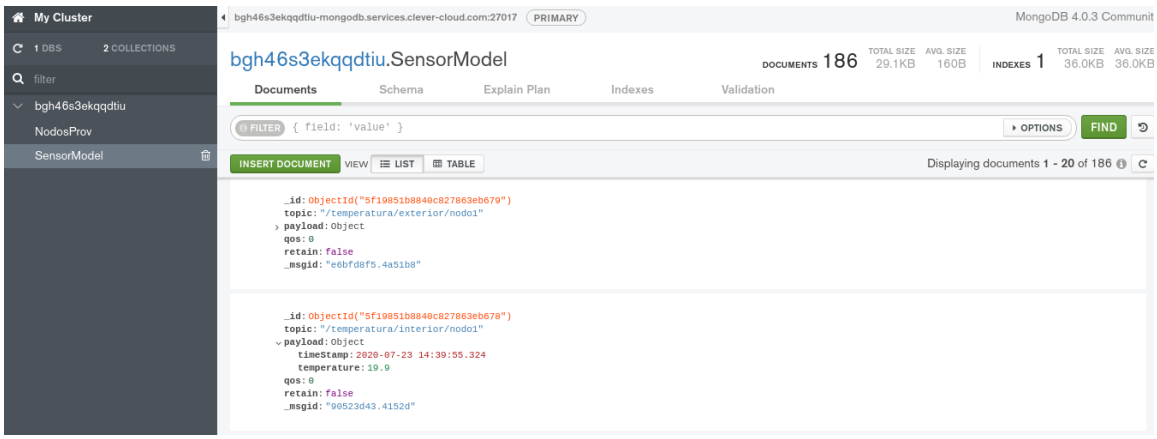

Figura 72. Colección SensorModel en MongoDB Compass

Finalmente, como se comentaba en secciones anteriores, también es posible que el panel de control no reporte los valores de temperatura de manera pasiva. Al disponer cada panel de cada nodo aprovisionado dos botones, se puede solicitar (pulsando estos) por MQTT al cliente el valor actual de temperatura deseado.

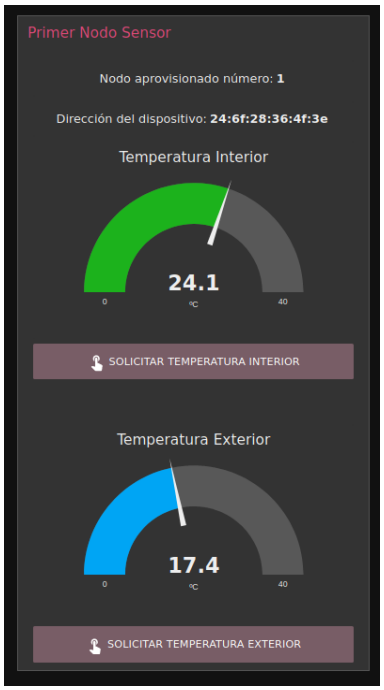

Figura 73. Panel de Control del primer nodo aprovisionado

Una vez se pulsa el botón de *Solicitar Temperatura Interior* desde el panel de control, el cliente recibe una nueva solicitud MQTT, y este le envía la temperatura sintética del nodo en cuestión.

```
I (1259875) MQTT_EXAMPLE: MQTT_EVENT_DATA
I (1259875) MQTT_EXAMPLE: Nueva solicitud:
TOPIC = /solicitud/tempInt
NODO = 1
I (1259875) MQTT_EXAMPLE: Confirmación Topic: /solicitud/tempInt
I (1259885) MQTT_EXAMPLE: Confirmación Nodo: 1
I (1259885) MQTT_EXAMPLE: Temperatura interior: 24.1 ºC
I (1259895) MQTT_EXAMPLE: sent publish successful...
I (1259905) MQTT_EXAMPLE: MQTT_CHECK
```

Figura 74. Solicitud y envío de la temperatura interior del nodo 1

Por otra parte, en la pestaña *Monitorización de la Temperatura*, el usuario puede visualizar los diferentes valores de temperatura que han tomado los sensores durante la última hora para así hacerse una idea de cómo de equilibradas están la temperatura interior y exterior y si hay algún punto en el que estas hayan variado drásticamente, lo cual puede indicar que hubo un error por parte del sensor o hay un problema real de temperatura.

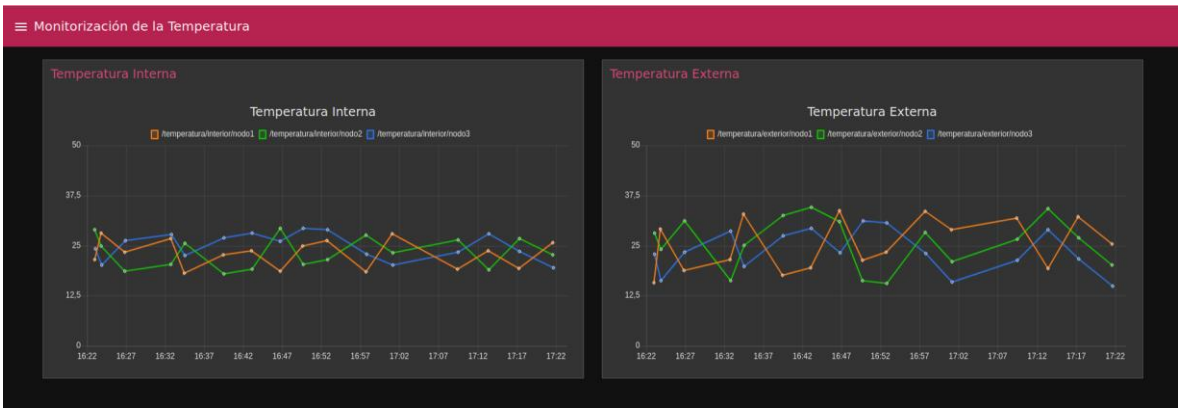

Figura 75. Valores de las temperaturas en determinados momentos de tiempo

Capítulo 7

Conclusiones y líneas de futuro

Después de haber profundizado en la tecnología Bluetooth Mesh, es el momento de analizar en este capítulo si los objetivos marcados al principio del proyecto se han cumplido con éxito. Para ello, se listan una serie de conclusiones en el siguiente apartado, y, además, se detallarán las principales líneas futuras que podrían añadir valor al trabajo.

7.1. Conclusiones

Como se vio en la definición de objetivos del capítulo 1 del presente proyecto, uno de los principales objetivos que había que cubrir era que este sirviese como una clara guía sobre Bluetooth Mesh. Al tratarse de una novedosa tecnología, conseguir esto requirió un gran trabajo de investigación, pues se necesitó toda la documentación oficial proporcionada por Bluetooth SIG, y, por supuesto, de formación personal sobre el tema, ya que así era la única manera de poder transmitir al lector los conceptos claves que diferencian a la malla Bluetooth de otros protocolos que implementan la topología de red en malla. Tras considerar el trabajo realizado, el objetivo se ha conseguido satisfactoriamente porque a lo largo de todo el presente documento se brinda una buena visión general de la tecnología Bluetooth Mesh, especialmente en conceptos como elementos y modelos, dirección de grupo, claves de aplicación, red y dispositivo, aprovisionamiento, el cual cuenta con un apéndice dedicado exclusivamente para él, etc.

Por otro lado, el segundo objetivo principal del proyecto consistía en comprobar que el framework ESP-IDF daba soporte a la malla Bluetooth tal como indicaba en su página oficial. Observando el capítulo 6 se obtienen las pruebas que hacen evidente que a través de este entorno de desarrollo es posible desarrollar software funcional sobre el SoC ESP32. Obviamente, conseguir esto supuso un esfuerzo en cuanto a instalación, configuración y comprensión del entorno, ya que los proyectos basados en ESP-IDF se desarrollan desde un terminal y esto al principio resulta más complejo que si se trabajase con un entorno más gráfico como puede ser Eclipse IDE.

En el capítulo 5 se deja constancia de que Bluetooth Mesh se puede usar en aplicaciones IoT como *Smart Buildings, Smart Home* o *Smart Lighting*, y que a día de hoy hay hasta 752 proyectos certificados por Bluetooth SIG que cumplen con la especificación de la malla Bluetooth. Desde luego, esto hace pensar que esta tecnología apunta fuerte en soluciones IoT a gran escala, por lo que gracias a los proyectos de ejemplo BLE Mesh que proporciona *Espressif* en su SDK, se pudo comenzar a desarrollar aplicaciones propias, para así ver esta tecnología en acción y comprobar la veracidad de sus ventajas, como el aumento del rango de cobertura o la interoperabilidad.

Inicialmente, resultó costoso entender de qué manera era necesario desarrollar el software, sin embargo, con esfuerzo y dedicación se llegó a comprender cómo se deberían estructurar los programas para que al *flashearlos* sobre, en este caso, las ESP32, dieran capacidad a estas

de funcionar de una determinada manera. De todos modos, conocer la teoría de la malla Bluetooth, sobre todo la parte de los modelos estándar, ayudó positivamente a la curva de aprendizaje. De hecho, gracias a los proyectos de ejemplo, se logró desarrollar un código propio (apartado 6.2. del capítulo 6), en el que se consiguió integrar en un mismo nodo, funcionando como modelo sensor, tres tecnologías diferentes, es decir, Bluetooth Mesh, Wi-Fi y MQTT. Añadir estos dos protocolos hizo de la aplicación algo más sofisticado, pues a través de tecnologías vistas durante el año de máster como Node-RED y MongoDB, se consiguió hacer un panel de control, desde el que el usuario recibía información de los nodos o les hacía llegar órdenes a través de MQTT, y una base de datos en la que se almacenaban la lista de los nodos aprovisionados de la red y los datos que estos, pues simulaban ser sensores, recogían.

A nivel personal, estoy satisfecha con el resultado final de este trabajo, pues además de haber adquirido nuevos conocimientos de una tecnología totalmente prometedora para el IoT de manera autodidacta, he logrado anteponerme a los baches que han ido surgiendo a lo largo de los meses de desarrollo y he conseguido demostrar a través de un par de pruebas de concepto la funcionalidad de Bluetooth Mesh.

7.2. Líneas de futuro

Tras comprobar que el proyecto cumple con los objetivos fijados, es el momento de dejar las puertas abiertas a mejoras o posibles cambios que ayuden a mejorar el trabajo propuesto. Por ello, a continuación, se listas las futuras líneas de desarrollo en caso de que se quiera ampliar el proyecto basado en la malla Bluetooth.

- En primer lugar, como se comentaba en la sección 3.4.2. del capítulo 3, la especificación de los modelos de malla de Bluetooth SIG define de manera rigurosa y extendida 4 grupos de modelos de malla estándar. En el proyecto se trabaja con los modelos genéricos y los modelos sensores, por ello se propone trabajar con algunos de los otros dos grupos para ver qué funcionalidades, a modo de pruebas de concepto, presentan.

- ESP-IDF, además de ofrecer ejemplos sobre Bluetooth Mesh, también proporciona ejemplos de uso de otras tecnologías para trabajar sobre los SoC ESP32. De hecho, se consiguió la coexistencia en un mismo nodo en la solución de sensorización desarrollada en este framework (apartado 6.2. del capítulo 6) porque se proporcionaba el material para hacerlo posible. En la solución desarrollada se usó MQTT, sin embargo, queda como propuesta implementar otros protocolos vistos durante el año de máster y muy presentes en el IoT, por ejemplo, CoAP.

- Otra idea futura sería trabajar con otra plataforma de desarrollo, por ejemplo, con la de *Nordic Semiconductor*, ya que, recordando, proporciona un kit de desarrollo denominado *nRF5 SDK for Mesh*, que incluye ejemplos e instrucciones claras sobre cómo construir una red o cómo crear nuevos modelos. Después de usarla se podría hacer una comparación entre este entorno de desarrollo y el de *Espressif* para

aportar más luz a los futuros desarrolladores de soluciones IoT que quieran implementar Bluetooth Mesh.

- A pesar de que se intentó por todos los medios usar *Zephyr*, sería interesante crear futuras pruebas de concepto con él cuando este ofrezca un mejor soporte para, en este caso, las placas basadas en SoC ESP32, puesto que promete ser un sistema operativo con mucho recorrido para IoT.

- Por último, y a modo de propuesta, partiendo de la base de que Bluetooth Mesh es una prometedora tecnología en aplicaciones IoT, este trabajo, que tiene una sólida parte teórica y práctica, se podría llevar al ámbito docente, en concreto a la asignatura de *Redes, Protocolos e Interfaces I*, pues en ella se tratan diversos protocolos de comunicación inalámbrica. Entre ellos destaca Bluetooth Low Energy, lo que hace interesante introducir a los alumnos la red en malla Bluetooth, con conceptos teóricos del capítulo 3 y una prueba sencilla de entender como puede ser la que se desarrolla en el apartado 6.1. del capítulo 6.

Chapter 8

Introduction

In recent years, the *Internet of Things* (IoT) is becoming more popular in society due to its boom and technological advancement. Although the precise definition of IoT today is almost unattainable, it can be defined as the paradigm in which computing and networking capabilities are integrated into any type of object with the aim of using them to query the status of said object and to change your status whenever possible. It could also be defined in a more simplified way as the ability of devices to communicate with each other over, for example, one or more wireless networks.

With the continuous expansion of the *Internet of Things*, more and more connected devices have started to be used, from mobile phones to autonomous vehicles or control systems. This causes the flexibility, agility and reliability of existing network architectures to be questioned, since it is questioned whether they will be able to cope with the increasing demands imposed on them by this vertiginous growth of new technologies.

In 2017, a Gartner[40] [1] predictive report said that the number of connected IoT devices in 2019 would be around 14 billion, while in 2021 it would exceed 25 billion. Three years later, a Business Insider[41] [2] research report has confirmed that these predictions outperformed reality; however, that does not take away from the current surprising prediction, because after having dealt with different companies and consumers, it is estimated that there will be more than 41 billion connected IoT devices by 2027, compared to 8 billion in 2019.

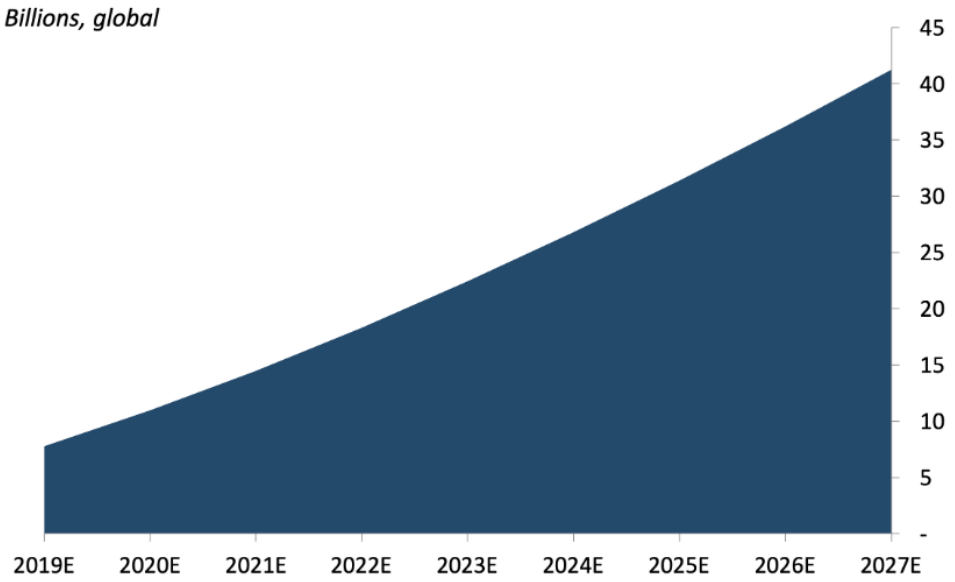

Figura 76. Prediction of connected IoT devices [2]

[40] *Gartner Inc.* is an information technology research and consulting company (*https://www.gartner.com/en*).
[41] *Business Insider* is an American digital business and financial news outlet, which also publishes technology. (*https://www.businessinsider.es/?r=US&IR=T*).

According to the data illustrated in Figure 76, in the coming years the number of IoT devices will grow exponentially, so it is necessary to look for an attractive communication option for anyone who wants to use a wide spectrum of these in their projects.

Architectures based on mesh networks (*mesh*) are a good option to accommodate the continuous waves of new IoT devices. A *mesh* network is nothing more than a network topology in which each node belonging to it is connected to all the other nodes in the network. This type of topology has a large number of benefits, among them, *reliability* in scenarios where connectivity is essential, because if any of the nodes fails, data transfer is not difficult due to their automatic routing. In addition, the larger the deployments, the less problems there will be with the network coverage distance, since it will be extended by allowing it to extend its reach.

Despite having these advantages, in the industry there is not only one wireless communication protocol that adopts this red topology, so it is necessary to determine which is the most appropriate based on the conditions and expectations of the project to be developed.

8.1. Motivation

In recent years, it is increasingly common to seek to satisfy in the area of communication technologies requirements such as *interoperability*, the possibility of covering *large areas*, reduction and optimization of *energy consumption*, ability to control and monitor a large *number of devices*, etc.

Within wireless communications, those that adopt *mesh* topologies are indicated to satisfy this series of requirements. Currently there is a wide variety of these, but some have problems related to low data rates, the limited number of "hops" when transmitting data through the mesh, scalability limits and difficulties in following the procedures to change the composition of the mesh network device.

The creation of a standard *mesh* network technology based on *Bluetooth Low Energy* (BLE) [3] gave IoT application developers the opportunity to meet the requested requirements, but without the associated limitations and restrictions, since, after all, interoperability and energy efficiency are the hallmarks of Bluetooth LE.

With the advent of Bluetooth Mesh, it is not necessary to add any additional hardware, as it can be applied to any BLE device whose version is 4.0 or higher. This makes it gather more points in favor, since as is known, Bluetooth LE has great compatibility with mobile phones, computers, tablets and other types of peripherals, such as smart bracelets, which satisfies another of the requirements.

Unlike Bluetooth BR/EDR, which only supported point-to-point connection (*1:1*) or Bluetooth Low Energy, which in addition to supporting point-to-point communication, had the ability to broadcast data (*1:m*), Bluetooth Mesh enables *many-to-many* (*m:m*) communication, which is ideal for IoT applications such as building automation, commercial lighting, asset tracking, or sensor networks, which require hundreds or thousands of devices to communicate reliably and securely.

8.2. Objectives

The main objective of this project will be double. First of all, Bluetooth Mesh will be described in detail, since after studying all the documentation that exists on this technology, a guide will be made that talks about both its operation and implementation and the security that surrounds it. Second, it will be verified that the ESP-IDF *framework* [3] supports Bluetooth Mesh. To make this possible, a series of *software* developments will be carried out with some specific use cases on the ESP32 SoC [4].

To achieve these main objectives, it is necessary to define secondary objectives, which will provide a solid base of knowledge for the development of the work. The list of these objectives will be:

- Analyze and document the current state of wireless communication technologies that implement the mesh network topology.
- Document the part of the Bluetooth Low Energy stack on which Bluetooth Mesh is based on.
- Study the different development tools that are provided in the market and that support BLE Mesh technology.
- Collect information about the selected SoC and the development board to later document in a clear and concise way.
- Analyze, study and document the different provisioning mechanisms for Bluetooth LE devices.
- Analyze IoT applications in which Bluetooth Mesh is used. Give some real example of these.
- Study some of the different models proposed by the Bluetooth community to start with application development.
- Development of proof-of-concept software dedicated to specific Bluetooth Mesh models.
- Development of a control panel using the Node-RED tool.
- Check the correct functioning of the systems implemented by simulating different scenarios.

8.3. Methodology

Methodology is understood as the set of procedures that allow us to achieve a specific objective. In this case, you want to achieve the list of goals mentioned above. For this, the work has been divided into three stages in order to achieve the greatest possible success in its development.

i. Investigation stage:

This first stage will be divided into two, as two different investigations will be carried out. Initially, it will begin by analyzing the current state of the different wireless communication technologies that support the mesh network topology. In this way, a well-founded idea of the pros and cons of each of them will be created, and, in addition, it will lead to the introduction of the new technology on which this project revolves, that is, Bluetooth Mesh. The second investigation will be linked to the different companies that claim to have development kits that support BLE Mesh. Through it, the different resources they offer will be analyzed, however, it will be up to the Bluetooth Mesh solutions developer himself to select, for example, the hardware platform with which he wishes to work.

ii. Study stage:

Like the previous stage, the study of work will take two different paths. You must first obtain a detailed description of Bluetooth Mesh, so it will be necessary to have all the official documentation. After studying it, new concepts can be added to this document with which a general idea of this technology will be gradually obtained. These concepts will include the term *provisioning*, the type of communication model used between nodes, security based on separation of concepts, etc. Afterwards, the study will be focused on the development environment on which a BLE Mesh solution will be implemented. To do this, the corresponding installation, configuration and programming guides in ESP-IDF must be followed.

iv. Development stage:

In this last stage it is intended to verify that the ESP-IDF development environment provides adequate support so that developers can successfully implement Bluetooth Mesh solutions in IoT applications. To do this, different proofs of concept will be created where different models of this network technology will be used. Aspects such as programming from scratch of a basic model will be explored to later verify how the advantages of a mesh topology are taken advantage of, such as, for example, the increase in the coverage area despite dealing with Bluetooth LE devices whose coverage range it is more reduced. Similarly, a more elaborate proof of concept will be developed in which several technologies will coexist to obtain a complete sensorization solution.

8.4. Memory structure

This document consists of nine chapters, which are briefly described below.

- The first chapter serves as an *introduction*, since in addition to presenting the project, the motivation is included, the objectives are raised and the methodology to be followed for its development is described.

- The second chapter introduces the *state of the art*, detailing the current wireless communications technologies that support mesh topology in their implementations.

- The third chapter is the basis of the project itself. It describes Bluetooth Mesh in detail so that the reader may have the necessary knowledge before proceeding to the following points.

- The fourth chapter describes the tools selected both *hardware* and *software* for the development of the project.

- The fifth chapter talks about possible IoT *applications* in which Bluetooth Mesh can be used, as well as solutions already developed and that have been successful.

- In the sixth chapter the *proofs of concept* carried out will be documented, which help to verify if ESP-IDF supports Bluetooth Mesh, and, therefore, successful results are achieved.

- The seventh chapter presents the *conclusions* of the project and some ideas for the future that can add value to the work.

- In the eighth and ninth chapters are the *introductory* chapter and *conclusions* chapter translated into English.

- Finally, it should be mentioned that, after the bibliography, there are a series of very relevant appendices that complete the reading of the work. In the first one, the first layers of the Bluetooth Low Energy protocol stack are covered, while the second appendix is an installation and configuration guide for the ESP-IDF development environment. Finally, the third appendix offers three real provisioning solutions.

Chapter 9

Conclusions and future lines

After having delved into Bluetooth Mesh technology, it is time to analyze in this chapter whether the objectives set at the beginning of the project have been successfully met. To do this, a series of conclusions are listed in the next section, and, in addition, the main future lines that could add value to the work will be detailed.

9.1. Conclusions

As seen in the definition of objectives in chapter 1 of this project, one of the main objectives to be covered was to serve as a clear guide on Bluetooth Mesh. Being a new technology, achieving this required a lot of research, since all the official documentation provided by the Bluetooth SIG was needed and, of course, personal training on the subject, since this was the only way to convey to the reader the key concepts that differentiate Bluetooth mesh from other protocols that implement the mesh network topology. After considering the work done, the goal has been successfully achieved because throughout this document a good overview of Bluetooth Mesh technology is provided, especially in concepts such as elements and models, group address, application keys, network and device, provisioning, which has an appendix dedicated exclusively to it, etc.

On the other hand, the second main objective of the project was to verify that the ESP-IDF framework supported the Bluetooth mesh as indicated on its official page. Observing chapter 6, the evidence is obtained that make it clear that through this development environment it is possible to develop functional software on the ESP32 SoC. Obviously, achieving this involved an effort in terms of installation, configuration and understanding of the environment, since projects based on ESP-IDF are developed from a terminal and this at the beginning is more complex than if you were working with a more graphical environment such as Eclipse IDE.

In chapter 5 it is stated that Bluetooth Mesh can be used in IoT applications such as *Smart Buildings*, *Smart Home* or *Smart Lighting*, and that today there are up to 752 projects certified by Bluetooth SIG that comply with the Bluetooth mesh specification. Of course, this suggests that this technology is targeting large-scale IoT solutions, so thanks to the BLE Mesh sample projects provided by Espressif in its SDK, it was possible to start developing its own applications, in order to see this technology in action and verify the veracity of its advantages, such as increasing the coverage range or interoperability.

Initially, it was costly to understand how it was necessary to develop the software, however, with effort and dedication it was come to understand how the programs should be structured so that when flashing them on, in this case, the ESP32, they would give them the ability to function in a certain way. Anyway, knowing the Bluetooth mesh theory, especially the part of the standard models, positively helped the learning curve. In fact, thanks to the example projects, it was possible to develop its own code (section 6.2. of

chapter 6), in which it was possible to integrate in the same node, functioning as a sensor model, three different technologies, that is, Bluetooth Mesh, Wi-Fi and MQTT. Adding these two protocols made the application somewhat more sophisticated, because through technologies seen during the master's year such as Node-RED and MongoDB, it was possible to create a control panel, from which the user received information from the nodes or sent them orders through MQTT, and a database in which the list of the network's provisioned nodes and data were stored that these, as they pretended to be sensors, collected.

Personally, I am satisfied with the final result of this work, because in addition to having acquired new knowledge of a totally promising technology for the IoT in a self-taught way, I have managed to overcome the bumps that have arisen throughout the months of development and I have managed to demonstrate through a couple of proofs of concept the functionality of Bluetooth Mesh.

9.2. Future lines

After verifying that the project meets the objectives set, it is time to leave the doors open to improvements or possible changes that help improve the proposed work. For this reason, the future lines of development are listed below in case you want to expand the project based on the Bluetooth mesh.

- First, as discussed in section 3.4.2. from chapter 3, the Bluetooth SIG mesh model specification rigorously and extensively defines 4 groups of standard mesh models. The project works with generic models and sensor models, therefore it is proposed to work with some of the other two groups to see what functionalities, by way of proofs of concept, they present.

- ESP-IDF, in addition to offering examples on Bluetooth Mesh, also provides examples of use of other technologies to work on ESP32 SoCs. In fact, coexistence was achieved in the same node in the sensorization solution developed in this framework (section 6.2. of chapter 6) because the material was provided to make it possible. In the developed solution, MQTT was used, however, the proposal remains to implement other protocols seen during the master's year and very present in the IoT, for example, CoAP.

- Another future idea would be to work with another development platform, for example, with that of *Nordic Semiconductor*, since, remembering, it provides a development kit called *nRF5 SDK for Mesh*, which includes examples and clear instructions on how to build a network or how to create new models. After using it, a comparison could be made between this development environment and *Espressif*'s to shed more light on future developers of IoT solutions who want to implement Bluetooth Mesh.

- Despite the fact that *Zephyr* was tried by all means, it would be interesting to create future proofs of concept with it when it offers better support for, in this case, ESP32 SoC-based boards, since it promises to be an operating system with a lot of journey for IoT.

- Finally, and as a proposal, starting from the basis that Bluetooth Mesh is a promising technology in IoT applications, this work, which has a solid theoretical and practical part, could be taken to the teaching field, specifically to the subject of *Networks, Protocols and Interfaces I*, since it deals with various wireless communication protocols. Among them, Bluetooth Low Energy stands out, which makes it interesting to introduce students to the Bluetooth mesh network, with theoretical concepts from chapter 3 and a simple to understand test such as the one developed in section 6.1. from chapter 6

Bibliografía

[1] Gupta, A., Tsai, T., Rueb, D., Yamaji, M., & Middleton, P. (2017). *Forecast: Internet of Things – Endpoints and Associated Services, Worldwide.* Gartner Research. Disponible en: https://www.gartner.com/en/documents/3840665.

[2] Newman, P. (2020). *THE INTERNET OF THINGS 2020: Here's what over 400 IoT decision-makers say about the future of enterprise connectivity and how IoT companies can use it to grow revenue.* Business Insider. Disponible en: https://www.businessinsider.com/internet-of-things-report?IR=T.

[3] Espressif: *ESP-IDF Programming Guide.* Disponible en: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/ .

[4] Espressif: *ESP32.* Disponible en: https://www.espressif.com/en/products/socs/esp32.

[5] Bluetooth SIG: Woolley, M. (2017). *An Intro to Bluetooth Mesh Part 1.* Disponible en: https://www.bluetooth.com/blog/an-intro-to-bluetooth-mesh-part1/.

[6] Bluetooth SIG: Woolley, M. (2017). *An Intro to Bluetooth Mesh Part 2.* Disponible en: https://www.bluetooth.com/blog/an-intro-to-bluetooth-mesh-part2/.

[7] Bluetooth SIG: Afaneh, M. (2020). *Wireless Connectivity Options fot IoT Applications – Terms and Applications.* Disponible en: https://www.bluetooth.com/blog/wireless-connectivity-options-for-iot-applications-terms-and-applications/?utm_campaign=range&utm_source=internal&utm_medium=blog&utm_content=wireless-connectivity-options-for-iot-applications-technology-comparison.

[8] Bluetooth SIG: Afaneh, M. (2020). *Wireless Connectivity Options fot IoT Applications – Technology Comparison.* Disponible en: https://www.bluetooth.com/blog/wireless-connectivity-options-for-iot-applications-technology-comparison/?utm_campaign=mesh&utm_source=internal&utm_medium=blog&utm_content=wireless-connectivity-options-for-iot-applications-condition-monitoring.

[9] Domoticalia. The smarthome experience. *¿Qué es Z-Wave?* Disponible en: https://www.domoticalia.es/es/content/14-z-wave.

[10] Develco Products. *Zigbee 3.0.* Disponible en: https://www.develcoproducts.com/technologies/zigbee/zigbee-30/.

[11] Kushalnagar, N., Montenegro, G., & Schumacher, C. (2007). *IPv6 over low-power wireless personal area networks (6LoWPANs): overview, assumptions, problem statement, and goals.* Disponible en: https://www.hjp.at/doc/rfc/rfc4919.html.

[12] Thread. *What is Thread?* Disponible en: https://www.threadgroup.org/What-is-Thread.

[13] LoRa Alliance. *What is the LoRaWAN Specification?* Disponible en: https://lora-alliance.org/about-lorawan.

[14] Pycom. *Pymesh (LoRa Mesh network created by Pycom).* Disponible en: https://docs.pycom.io/firmwareapi/pycom/network/lora/pymesh/.

[15] García, R. (2020). *¿Qué es WiFi Mesh? La solución para tener conexión en toda tu casa.* AZ adsl zone. Disponible en: https://www.adslzone.net/reportajes/tecnologia/que-es-wifi-mesh/.

[16] Bluetooth SIG: Kolderup, K. (2017). *Introducing Bluetooth Mesh Networking*. Disponible en: https://www.bluetooth.com/blog/introducing-bluetooth-mesh-networking/.

[17] Bluetooth SIG: Woolley, M. (2019). *Bluetooth mesh networking*. Disponible en: https://www.bluetooth.com/wp-content/uploads/2019/03/Mesh-Technology-Overview.pdf.

[18] Townsend, K., Cufí, C., & Davidson, R. (2014). *Getting started with Bluetooth low energy: tools and techniques for low-power networking*. " O'Reilly Media, Inc.".

[19] *Bluetooth MESH.* RTONE IoT Makers. Disponible en: https://rtone.fr/en/bluetooth-mesh-2/.

[20] *Diseño de aplicaciones inteligentes con bluetooth de bajo consumo con Bluetooth Mesh: Parte 1.* Colaboración de Editores de Digi-Key de América del Norte. (2018). Disponible en: https://www.digikey.es/es/articles/designing-bluetooth-low-energy-smart-applications-part-1.

[21] Bluetooth SIG: Mesh Networking Specifications. *Mesh Profile Specification 1.0.1.* Descarga disponible en: https://www.bt-stage.systems/specifications/mesh-specifications/.

[22] Bluetooth SIG: Mesh Networking Specifications. *Mesh Model Specification 1.0.1.* Descarga disponible en: https://www.bt-stage.systems/specifications/mesh-specifications/.

[23] Bluetooth SIG: Woolley, M. (2019). *Bluetooth Mesh Models. Technical Overview.* Disponible en: https://www.bluetooth.com/wp-content/uploads/2019/04/1903_Mesh-Models-Overview_FINAL.pdf.

[24] Bluetooth SIG: Ren, K., Woolley, M. (2017). *Bluetooth Mesh Security Overview.* Disponible en: https://www.bluetooth.com/blog/bluetooth-mesh-security-overview/.

[25] Bluetooth SIG: Ren, K. (2017). *Provisioning a Bluetooth Mesh Network Part 1.* Disponible en: https://www.bluetooth.com/blog/provisioning-a-bluetooth-mesh-network-part-1/.

[26] Bluetooth SIG: Ren, K. (2017). *Provisioning a Bluetooth Mesh Network Part 2.* Disponible en: https://www.bluetooth.com/blog/provisioning-a-bluetooth-mesh-network-part-2/.

[27] Ren, K. (2018). Bluetooth SIG: *3 things to know before choosing yout Bluetooth mesh hardware platform.* Descarga disponible en: https://www.bluetooth.com/bluetooth-resources/3-things-to-know-before-choosing-your-bluetooth-mesh-hardware-platform/.

[28] Silicon Labs: *Silicon Labs Bluetooth Mesh Software.* https://www.silabs.com/products/development-tools/software/bluetooth-low-energy/ble-mesh.

[29] Silicon Labs: *Simplicity Studio Software.* https://www.silabs.com/products/development-tools/software/simplicity-studio.

[30] Silicon Labs: *App Android: Bluetooth Mesh by Silicon Labs.* https://play.google.com/store/apps/details?id=com.siliconlabs.bluetoothmesh&hl=es_419.

[31] Silicon Labs: *App iOS: Bluetooth Mesh by Silicon Labs.* https://apps.apple.com/us/app/bluetooth-mesh-by-silicon-labs/id1411352948.

[32] Silicon Labs: *EFR32xG21 Bluetooth Starter Kit.* https://www.silabs.com/development-tools/wireless/efr32xg21-bluetooth-starter-kit.

[33] Nordic Semiconductor: *nRF5 SDK for Mesh.* https://www.nordicsemi.com/Software-and-tools/Software/nRF5-SDK-for-Mesh.

[34] Nordic Semiconductor: *Development kit nRF52 DK.* https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF52-DK.

[35] Nordic Semiconductor: *App Android: nRF Mesh.* https://play.google.com/store/apps/details?id=no.nordicsemi.android.nrfmeshprovisioner.

[36] Nordic Semiconductor: *App iOS: nRF Mesh.* https://apps.apple.com/us/app/nrf-mesh/id1380726771.

[37] STMicroelectronics: *STSW-BNRG-Mesh (Mesh over Bluetooth Low Energy).* https://www.st.com/en/embedded-software/stsw-bnrg-mesh.html#get-software.

[38] STMicroelectronics: *STEVAL-IDB008V2 (Platform base don the BlueNRG-2).* https://www.st.com/en/evaluation-tools/steval-idb008v2.html#sample-buy.

[39] STMicroelectronics: *App Android: ST BLE Mesh.* https://play.google.com/store/apps/details?id=com.st.bluenrgmesh&hl=es_419.

[40] STMicroelectronics: *App iOS: ST BLE Mesh.* https://apps.apple.com/ca/app/bluenrg-mesh/id1348645067?ign-mpt=uo%3D4.

[41] Espressif: *ESP-BLE-MESH.* Disponible en: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/esp-ble-mesh/ble-mesh-architecture.html.

[42] Espressif: *ESP32-DevKitC overview.* Disponible en: https://www.espressif.com/en/products/devkits/esp32-devkitc/overview.

[43] *Zephyr Project Brings IoT Expertise to the Embedded Linux Conference Europe.* (2017). Disponible en: https://www.zephyrproject.org/zephyr-project-brings-iot-expertise-to-the-embedded-linux-conference-europe/.

[44] Zephyr Project: *Supported Board.* Disponible en: https://docs.zephyrproject.org/latest/boards/index.html#boards.

[45] Bluetooth SIG: *An Introduction to Bluetooth Mesh Software Development.* (2020). Descarga disponible en: https://www.bluetooth.com/bluetooth-resources/bluetooth-mesh-developer-study-guide/.

[46] *ESP32* (imagen diagrama de funciones del ESP32). Wikipedia. Disponible en: https://es.wikipedia.org/wiki/ESP32.

[47] Espressif: *ESP32-DevKitC V4 Getting Started Guide.* https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html.

[48] *ESP-WROOM-32 DEV KIT MODULE.* Disponible en: https://www.flickr.com/photos/jgustavoam/40089095211.

[49] Espressif: *Build and flash with Eclipse IDE.* https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/eclipse-setup.html.

[50] Espressif: *Espressif IoT Development Framework.* Disponible en: https://github.com/espressif/esp-idf/tree/release/v4.2.

[51] Espressif: *Get Started.* Disponible en: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html.

[52] Bluetooth SIG: Marcel, J. (2020). *Bluetooth Emerging Market Forecasts Predict Continued Growth Through 2024.* Disponible en: https://www.bluetooth.com/blog/bluetooth-emerging-market-forecasts-predict-continued-growth-through-2024/.

[53] Bluetooth SIG: Marcel, J. (2019). *New Location Services Use Cases Enhance Building Operational Efficiencies*. Disponible en: https://www.bluetooth.com/blog/new-location-services-usecases-enhance-buildingoperational-efficiencies/?utm_campaign=bmu&utm_source=internal&utm_medium=blog&_content=bluetooth-emerging-market-forecasts-predict-continued-growth-through-2024.

[54] Bluetooth SIG: Marcel, J. (2019). *Bluetooth is Getting Precise with Positioning Systems.* Disponible en: https://www.bluetooth.com/blog/bluetooth-positioning-systems/?utm_campaign=location-services&utm_source=internal&utm_medium=blog&utm_content=new-location-services-use-cases-enhance-building-operational-efficiencies.

[55] Bluetooth SIG: *Smart Building.* Disponible en: https://www.bluetooth.com/learn-about-bluetooth/markets/smart-building/.

[56] Bluetooth SIG: *Qualify Your Product.* Disponible en: https://www.bluetooth.com/develop-with-bluetooth/qualification-listing/ .

[57] Bluetooth SIG: *Qualified Mesh Products.* Disponible en: https://www.bluetooth.com/learn-about-bluetooth/bluetooth-technology/topology-options/le-mesh/mesh-qualified/.

[58] Bluetooth SIG: Hollander, D. (2018). *Bluetooth Mesh – What a Difference a Year Makes.* Disponible en: https://www.bluetooth.com/blog/what-a-year-for-bluetooth-mesh/.

[59] Evangeline, H. (2018). *LEDVANCE Introduces World's First Bluetooth Mesh Qualified LED Products*. Disponible en: https://www.ledinside.com/news/2018/1/ledvance_introduces_worlds_first_bluetooth_mesh_qualified_led_products.

[60] Bluetooth SIG: Slupik, S. (2020). *The Myths, Facts, and Future of Wireless Lighting Control.* Disponible en: https://www.bluetooth.com/blog/the-myths-facts-and-future-of-wireless-lighting-control/.

[61] SILVAIR: *Retrofitting the Brussels office of Macq.* Disponible en: https://silvair.com/media/filer_public/01/41/0141da06-c97e-4f6f-b454-6d6345e77a6d/2019_retrofitting_the_brussels_office_of_macq_rev1.pdf.

[62] SILVAIR: *Bluetooth mesh standard reaches new heights*. Disponible en: https://silvair.com/media/filer_public/b2/a1/b2a1ea33-304e-4d65-b1b6-a572c79a8ee7/bluetooth_mesh_standard_reaches_new_heights_-_retrofitting_lighting_in_the_stratospheretower_case_study.pdf.

[63] SILVAIR: *Bluetooth mesh standard enters OSRAM office spaces*. Disponible en: https://silvair.com/media/filer_public/3c/15/3c1580f7-e052-42b1-a47d-833484e86555/bluetooth_mesh_standard_enters_osram_office_spaces-_office_retrofit_with_minimal_disruption_case_study_r07.pdf.

[64] SILVAIR: *Lighting up Tamaha warehouse with Bluetooth mesh standard.* Disponible en: https://silvair.com/media/filer_public/33/9a/339a41ca-a2d0-48a0-a1f4-0fdb1281de40/lighting_controls_for_industry_40-_lighting_up_yamaha_warehouse_with_bluetooth_mesh_standard.pdf .

[65] Bluetooth SIG: *Developer Study Guide: Deploying BlueZ v5.50 on Raspberry Pi 3. Part 1 – Deployment.* Disponible en: https://3pl46c46ctx02p7rzdsvsg21-wpengine.netdna-ssl.com/wp-content/uploads/2019/03/T1804_How-to-set-up-BlueZ_LFC_FINAL-1.pdf?utm_campaign=developer&utm_source=internal&utm_medium=blog&utm_content=bluez-on-pi3.

[66] Bluetooth SIG: *Step-by-Step Guide. How to Deploy BlueZ v5.50 on Raspberry Pi 3 and Use It. Part 2 – Provisioning*. Disponible en: https://3pl46c46ctx02p7rzdsvsg21-wpengine.netdna-ssl.com/wp-content/uploads/2019/03/Tutorial-How-to-set-up-BlueZ_Part2-3.pdf.

[67] Bluetooth SIG: *How to Deply BlueZ on a Raspberry Pi Board as a Bluetooth Mesh Provisioner.* Descarga disponible en: https://www.bluetooth.com/bluetooth-resources/developer-study-guide-how-to-deploy-bluez-on-a-raspberry-pi-board-as-a-bluetooth-mesh-provisioner/ .

[68] Zephyr: *X86 Emulation (QEMU).* Disponible en: https://docs.zephyrproject.org/2.2.0/boards/x86/qemu_x86/doc/index.html.

[69] Zephyr: *Using BlueZ with Zephyr.* Disponible en: https://docs.zephyrproject.org/2.2.0/guides/bluetooth/bluetooth-tools.html?#bluetooth-bluez.

Apéndices

Apéndice I. Bluetooth Low Energy

Bluetooth Low Energy (BLE), también conocido como Bluetooth Smart, fue introducido en 2010 por Bluetooth SIG como parte de la especificación básica Bluetooth 4.0. Aunque existen ciertas características en común entre la especificación clásica de Bluetooth (BR/EDR) y BLE, este último apareció con el objetivo de corregir antiguos errores de seguridad y para ampliar el uso de esta tecnología inalámbrica en dispositivos utilizados en IoT, los cuales se caracterizan por requerir un bajo consumo y una baja transmisión de datos.

La pila de Bluetooth Low Energy se divide en tres bloques principales (*Application, Host* y *Controller*), y a su vez, como se muestra en la Figura 77, cada bloque está compuesto por diferentes capas.

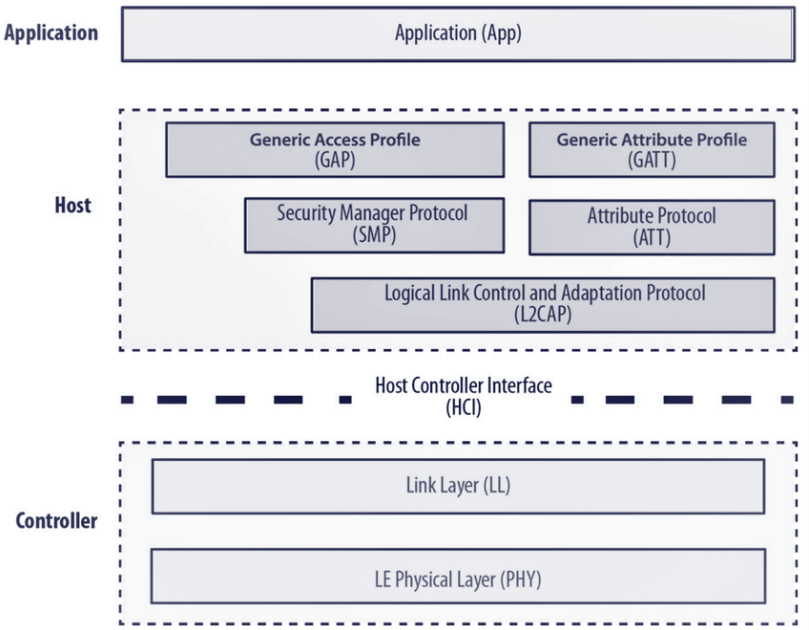

Figura 77. Pila del protocolo BLE

Para el propósito de este proyecto, solo se van a describir las capas pertenecientes al bloque *Controller*, es decir, la capa física y la de enlace, puesto que son las únicas que conserva la pila del protocolo Bluetooth Mesh. Los lectores interesados pueden consultar el libro *Getting started with Bluetooth low energy: tools and techniques for low-power networking* [18], el cual explica la funcionalidad de cada capa en detalle.

- Physical Layer (PHY):

  La capa física contiene la circuitería de comunicaciones que es capaz de realizar los procesos de modulación y demodulación de señales analógicas para posteriormente transformarlas en símbolos digitales. BLE opera en el mismo rango de frecuencias que la implementación clásica de Bluetooth (2,4 - 2,48 Ghz), pero utilizando otro conjunto de canales. En lugar de los 79 canales de 1 MHz, BLE usa 40 canales de 2 MHz. Dentro de un canal, los datos se transmiten utilizando

modulación por desplazamiento de frecuencia gaussiana (GFSK), con un *bitrate* de 1 Mbit/s y una potencia máxima de transmisión de 10 mW. En la siguiente imagen se puede observar la distribución de los canales de BLE.

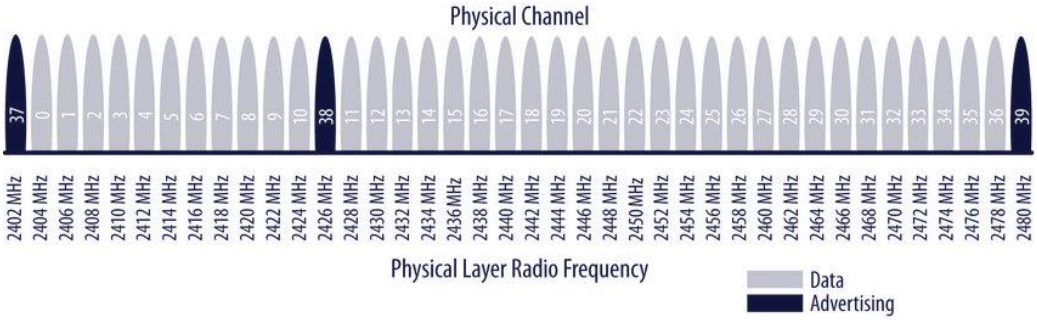

Figura 78. Canales de frecuencia

- Link Layer (LL):

Esta capa interactúa directamente con la capa física y se encarga de gestionar cómo se conectan los dispositivos unos con otros. Además, define los diferentes roles que puede desempeñar un dispositivo en el proceso de comunicación, es decir, *master, slaver, advertiser* y *scanner*.

El funcionamiento de esta capa viene definido por una máquina de estados, la cual se explica de forma detallada a continuación.

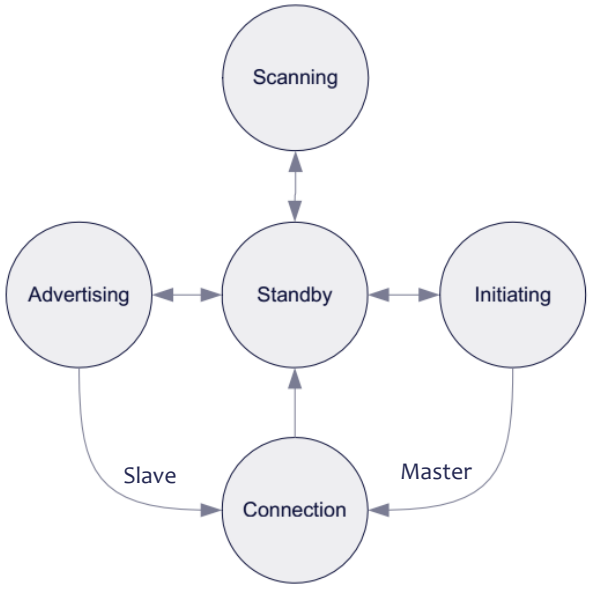

Figura 79. Máquina de estados

Como se puede ver en la figura 79, el estado central es el estado *Standby*, en el que se inicia la capa de enlace de cualquier dispositivo. En este estado, no se transmiten paquetes y tampoco se reciben. Por otro lado, un dispositivo *advertiser,* pasa a estar en estado *Advertising* cuando envía paquetes de datos de difusión (*broadcast*) con el fin de anunciarse y poder ser descubierto para establecer o no una conexión. Por el contrario, se estará en el estado de *Scanning* cuando un dispositivo *scanner* (*master*) inicia el proceso de búsqueda de otros dispositivos (*slavers*) que se encuentran en su área de alcance. En general, cuando se quiere establecer una

conexión con otro dispositivo, primero se debe estar en el estado *Initiating*, en el que se escucha al dispositivo al que se desea conectar. En el momento en el que el *master* recibe el paquete de anuncio del *slave*, se envía una solicitud de conexión y es entonces cuando se avanza al estado de *Connection*, el cual se puede considerar como el estado final al que se accede una vez que se ha pasado por los otros estados.

Apéndice II. Instalación de ESP-IDF

Este apéndice está destinado a explicar de manera breve los distintos pasos a seguir, incluyendo los correspondientes comandos a ejecutar, para conseguir la correcta instalación de ESP-IDF dentro del sistema operativo Ubuntu. Para ampliar la información que se muestra a continuación, se puede acceder a la página oficial de *Espressif* [51] en la que se ofrecen varias guías sobre la instalación, configuración y programación en ESP-IDF para diferentes sistemas operativos e incluso para diferentes versiones de este entorno de desarrollo.

i. Instalar requisitos previos

Para poder compilar con ESP-IDF se necesitan obtener los siguientes paquetes.

```
sudo apt-get install git wget flex bison gperf python python-pip python-
setuptools cmake ninja-build ccache libffi-dev libssl-dev dfu-util
```

Cabe destacar que se requiere una versión de *CMake* igual o posterior a la 3.5 para usar con éxito ESP-IDF. En mi caso, cuento con la versión 3.17.2.

ii. Obtener el repositorio ESP-IDF

Es importante contar con las bibliotecas de software proporcionadas por *Espressif* a través del repositorio en GitHub de ESP-IDF para poder crear aplicaciones para las ESP32. Por ello, es necesario clonar el repositorio en un directorio, por ejemplo, ***~/esp***, el cual actuará de directorio de instalación.

```
cd ~/esp
git clone -b release/v4.2 --recursive https://github.com/espressif/esp-
idf.git
```

Ahora, la versión 4.2 de ESP-IDF estará descargada en el directorio ***~/esp/esp-idf***. En caso de querer descargar la última versión o de actualizar la ya instalada, bastaría con quitar *-b release/v4.2* del comando anteriormente citado. Quedaría como:

```
git clone --recursive https://github.com/espressif/esp-idf.git
```

Como se verá a continuación, los programas de la cadena de herramientas usan variables de entorno *IDF_PATH* para acceder a ESP-IDF. Esta variable debe establecerse en el PC manualmente cada vez que este se inicia o establecer de forma permanente. En cualquier caso, se establece como: ***export IDF_PATH=~/esp/esp-idf***.

iii. Configuración de las herramientas

La versión de Linux de la cadena de herramientas (*toolchain*) se puede descargar desde el sitio web de *Espressif*: *https://dl.espressif.com/dl/xtensa-esp32-elf-linux64-1.22.0-80-g6c4433a-5.2.0.tar.gz*.

Para usar la cadena de herramientas, también se debe determinar una variable de entorno. En mi caso fue:
export ESPRESSIF_TOOLCHAIN_PATH=/home/lidia/.espressif/tolos/xtensa-esp32-elf/esp-2019r2-8.2.0/xtensa-esp32-elf.

A continuación, se instalan las herramientas utilizadas por este framework, como pueden ser el compilador, el depurador, los paquetes de Python, etc.

```
cd ~/esp/esp-idf
./install.sh
```

iv. Configurar las variables de entorno

Tras ejecutar el comando de instalación del paso anterior, este devuelve el comando necesario para configurar las variables de entorno (anteriormente declaradas) y así poder usar las herramientas ya instaladas en el terminal desde el que se va a trabajar.

```
~/esp/esp-idf/. ./export.sh
```

v. Iniciar un proyecto

Llegado este punto, el entorno ya estaría preparado para crear una aplicación. En el directorio ***~/esp/esp-idf/examples*** hay una gran variedad de ejemplos de los que partir para crear nuestros propios proyectos. En esta guía de instalación, se podría empezar con el famoso proyecto denominado *hello_world* (***~/esp/esp-idf/examples/get-started/hello_world***).

vi. Conexión del dispositivo

A continuación, se debe conectar el dispositivo al ordenador y comprobar qué puerto serie utiliza la ESP32. Al trabajar con Linux, el puerto serie comenzará con el patrón */dev/tty*.

vii. Configuración del proyecto

Una vez conocido el puerto serie ocupado por la placa (*/dev/ttyUSB0*), se accede al directorio del proyecto con el que se va a trabajar y se ejecuta la funcionalidad que ofrece ESP-IDF para poder configurar el proyecto.

```
~/esp/esp-idf/examples/get-started/hello_world
idf.py menuconfig
```

Tras la ejecución del último comando, se despliega un menú como el de la figura 80. Se puede navegar fácilmente por él para configurar las características del proyecto en cuestión. Por ejemplo, el nombre y contraseña de la red Wi-Fi o la velocidad del procesador.

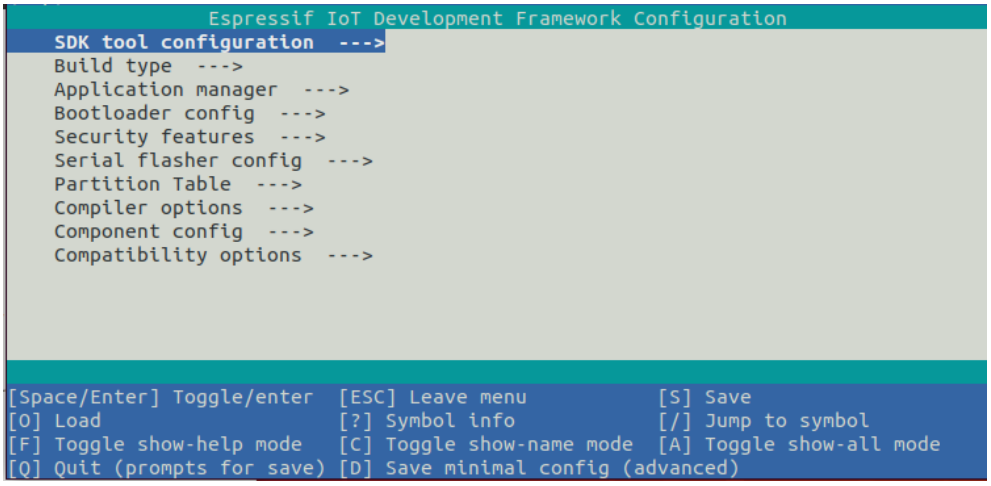

Figura 80. Pantalla de inicio del menú de configuración

Como se comentaba, la navegación por el menú es muy intuitiva. Las teclas de *flecha arriba* y *abajo* sirven para navegar por los diferentes submenús. La tecla *enter* permite entrar en dichos submenús, y la tecla *escape* permite salir de estos. También una tecla importante sería la *barra espaciadora*, pues habilita o deshabilita los elementos de la configuración a través de las casillas de verificación (**[*]**).

viii. Construcción del proyecto

Para la construcción del proyecto se debe ejecutar el comando que se muestra a continuación. Este compilará la aplicación y todos los componentes ESP-IDF. Además, generará un gestor de arranque, la tabla de particiones y los binarios de la aplicación.

```
idf.py build
```

Cuando la compilación finalice, se crea el archivo binario del firmware. En este proyecto este archivo sería *hello_world.bin*.

ix. Flashear en el dispositivo

Para ejecutar los binarios que se acaban de construir en la placa ESP32 que está conectada por puerto serie al ordenador se ejecuta:

```
idf.py -p PORT [-b BAUD] flash
```

Donde *PORT* es */dev/ttyUSB0* y *[-b BAUD]* es prescindible, pues la velocidad de transmisión predeterminada es de 115200 bps.

Al usar la opción *flash*, se crea y actualiza automáticamente el proyecto, por lo que no será necesario ejecutar de nuevo *idf.py build.*

Por otro lado, hay que presionar el botón *BOOT* de la ESP32 durante la carga del firmware para que este se cargue correctamente en el dispositivo.

x. Monitorizar

Finalmente, para verificar que, tras el proceso de *flash*, la placa inicia la aplicación *hello_world*, se monitoriza para poder ver lo que el proyecto con el que se está trabajando muestra por pantalla.

```
idf.py -p PORT monitor
```

Como se puede ver en la figura 81, la aplicación imprime *Hello world!* y a los 10 segundos se reinicia.

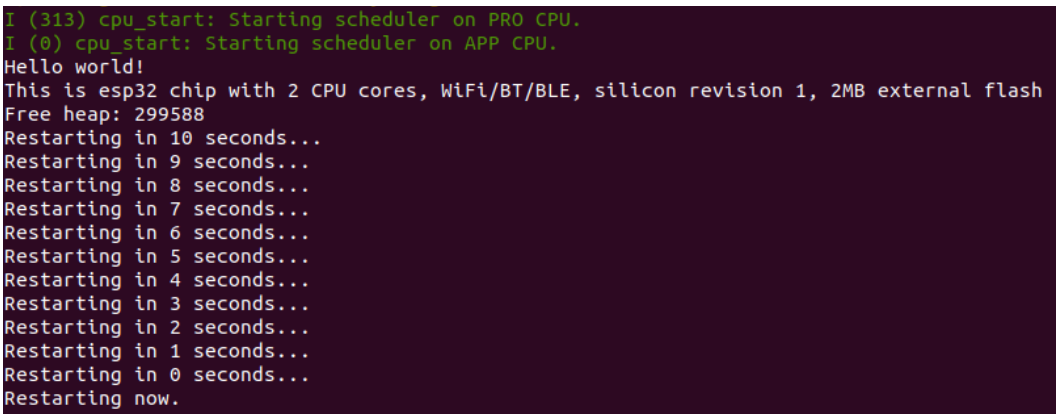

Figura 81. Monitorización de la placa que ejecuta el programa *hello_world*

Por último, para salir del monitor IDF habrá que pulsar de manera simultánea: ***ctrl+ alt gr +]***.

Apéndice III. Soluciones reales para aprovisionamiento

El objetivo de este segundo apéndice es aportar al lector otras soluciones de aprovisionamiento además de la descrita en el apartado 6.1. del capítulo 6 de este trabajo. A pesar de que la aplicación *nRF Mesh* desarrollada por *Nordic Semiconductor* es una buena manera para introducirse en este proceso de creación y configuración de redes Bluetooth Mesh, hay otras alternativas que pueden ajustarse mejor a las necesidades de otro tipo de proyecto.

A. Dispositivo BLE Mesh como aprovisionador

Hasta ahora no se había pensado que un nodo de una determinada red *mesh* (en este caso, las *ESP32-DevKitC V4*) pudiese funcionar como aprovisionador. Al hablar de administrador de red, siempre se había pensado en teléfonos móviles, tabletas u ordenadores, sin embargo, en Bluetooth Mesh cabe la posibilidad de que el propio pequeño dispositivo Bluetooth Low Energy realice las tareas de aprovisionamiento.

Este tipo de solución de aprovisionamiento sería beneficiosa, por ejemplo, para despliegues BLE Mesh en lugares remotos donde al usuario no le resulte fácil llegar hasta ellos para gestionar la unión de nuevo nodos a su red. Es por este motivo que ESP-IDF da el soporte necesario para hacer esto posible a través de un código de ejemplo localizable en el directorio *esp-idf/examples/bluetooth/esp_ble_mesh/ble_mesh_provisioner* del GitHub oficial de este framework.

En primer lugar, se realiza la inicialización de BLE Mesh, donde se llama a la función *esp_ble_mesh_provisioner_prov_enable(ESP_BLE_MESH_PROV_ADV | ESP_BLE_MESH_PROV_GATT)* para buscar dispositivos no aprovisionados en el entorno del dispositivo aprovisionador.

```c
static esp_err_t ble_mesh_init(void)
{
    uint8_t match[2] = {0xdd, 0xdd};
    esp_err_t err;

    prov_key.net_idx = ESP_BLE_MESH_KEY_PRIMARY;
    prov_key.app_idx = APP_KEY_IDX;
    memset(prov_key.app_key, APP_KEY_OCTET, sizeof(prov_key.app_key));

    esp_ble_mesh_register_prov_callback(example_ble_mesh_provisioning_cb);
    esp_ble_mesh_register_config_client_callback(example_ble_mesh_config_client_cb);
    esp_ble_mesh_register_generic_client_callback(example_ble_mesh_generic_client_cb);


    err = esp_ble_mesh_init(&provision, &composition);
    if (err) {
        ESP_LOGE(TAG, "Initializing mesh failed (err %d)", err);
        return err;
    }

    esp_ble_mesh_provisioner_set_dev_uuid_match(match, sizeof(match), 0x0, false);

    esp_ble_mesh_provisioner_prov_enable(ESP_BLE_MESH_PROV_ADV | ESP_BLE_MESH_PROV_GATT);

    esp_ble_mesh_provisioner_add_local_app_key(prov_key.app_key, prov_key.net_idx, prov_key.app_idx);

    ESP_LOGI(TAG, "BLE Mesh Provisioner initialized");

    return err;
}
```

Figura 82. Inicialización de BLE Mesh

Los dispositivos alcanzados por el aprovisionador, antes de ser aprovisionados automáticamente (Figura 84) por la pila del protocolo Bluetooth Mesh, se agregan a una lista de dispositivos denominada *nodes*. Al ser de tipo estructura (*esp_ble_mesh_node_info_t*), en esta lista no solo se almacenará durante el proceso de aprovisionamiento las MACs que identifican a los nodos, sino información relevante como el *uuid*, la dirección *unicast*, etc.

```c
typedef struct {
    uint8_t  uuid[16];
    uint16_t unicast;
    uint8_t  elem_num;
    uint8_t  onoff;
} esp_ble_mesh_node_info_t;

static esp_ble_mesh_node_info_t nodes[CONFIG_BLE_MESH_MAX_PROV_NODES] = {
    [0 ... (CONFIG_BLE_MESH_MAX_PROV_NODES - 1)] = {
        .unicast = ESP_BLE_MESH_ADDR_UNASSIGNED,
        .elem_num = 0,
        .onoff = LED_OFF,
    }
};
```

Figura 83. Almacenamiento de los nodos aprovisionados

Como se puede ver en la figura 83, el tamaño de esta lista de nodos que se pueden aprovisionar lo determina *CONFIG_BLE_MESH_MAX_PROV_NODES*. Si, por ejemplo, el valor de esta variable es 5, significa que el aprovisionador podrá aprovisionar hasta 5 dispositivos no aprovisionados. En teoría, un aprovisionador sin la limitación de su memoria puede aprovisionar hasta 32766 dispositivos, sin embargo, para limitar la memoria utilizada por un aprovisionador se podría poner un número máximo de 100. Por tanto, cuanto mayor sea ese valor, más le costará al aprovisionador almacenar la información de los nodos en su memoria.

```c
static esp_err_t prov_complete(int node_idx, const esp_ble_mesh_octet16_t uuid,
                               uint16_t unicast, uint8_t elem_num, uint16_t net_idx)
{
    esp_ble_mesh_client_common_param_t common = {0};
    esp_ble_mesh_cfg_client_get_state_t get_state = {0};
    esp_ble_mesh_node_info_t *node = NULL;
    char name[11] = {0};
    int err;

    ESP_LOGI(TAG, "node index: 0x%x, unicast address: 0x%02x, element num: %d, netkey index: 0x%02x",
             node_idx, unicast, elem_num, net_idx);
    ESP_LOGI(TAG, "device uuid: %s", bt_hex(uuid, 16));

    sprintf(name, "%s%d", "NODE-", node_idx);
    err = esp_ble_mesh_provisioner_set_node_name(node_idx, name);
    if (err) {
        ESP_LOGE(TAG, "%s: Set node name failed", __func__);
        return ESP_FAIL;
    }

    err = example_ble_mesh_store_node_info(uuid, unicast, elem_num, LED_OFF);
    if (err) {
        ESP_LOGE(TAG, "%s: Store node info failed", __func__);
        return ESP_FAIL;
    }

    node = example_ble_mesh_get_node_info(unicast);
    if (!node) {
        ESP_LOGE(TAG, "%s: Get node info failed", __func__);
        return ESP_FAIL;
    }

    example_ble_mesh_set_msg_common(&common, node, config_client.model, ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_GET);
    get_state.comp_data_get.page = COMP_DATA_PAGE_0;
    err = esp_ble_mesh_config_client_get_state(&common, &get_state);
    if (err) {
        ESP_LOGE(TAG, "%s: Send config comp data get failed", __func__);
        return ESP_FAIL;
    }

    return ESP_OK;
}
```

Figura 84. Aprovisionamiento automático de los nodos

Por otra parte, cuando un dispositivo BLE funciona como aprovisionador, también debe configurar parámetros como la *AppKey* o el *TTL* una vez finalizado el aprovisionamiento. La configuración correcta de las *AppKeys* es de vital importancia, ya que la aplicación solo puede enviar o recibir mensajes, es decir, establecer un estado u obtenerlo, después de que se haya establecido y vinculado la *AppKey* correspondiente.

Cuando el proceso de aprovisionamiento se completa con éxito, se llama a la función *esp_ble_mesh_provisioner_add_local_app_key* (Figura 82), que permite establecer y vincular una *AppKey*. Sin embargo, aquí no acaba el proceso de la configuración y vinculación de la *AppKey*, ya que el modelo que hace posible esto es el conocido como *Configuration Client*.

El proceso requerido para registrar el modelo *Configuration Client*, cuya función básicamente es representar un elemento que puede controlar y monitorear la configuración de un nodo, es similar al registro de otros modelos. En este código de ejemplo, además de registrarse el modelo *Configuration Client*, se registra el modelo obligatorio *Configuration Server* y el modelo *Generic On/Off Client*.

```c
static esp_ble_mesh_model_t root_models[] = {
    ESP_BLE_MESH_MODEL_CFG_SRV(&config_server),
    ESP_BLE_MESH_MODEL_CFG_CLI(&config_client),
    ESP_BLE_MESH_MODEL_GEN_ONOFF_CLI(NULL, &onoff_client),
};

static esp_ble_mesh_elem_t elements[] = {
    ESP_BLE_MESH_ELEMENT(0, root_models, ESP_BLE_MESH_MODEL_NONE),
};
```

Figura 85. Definición del modelo y del elemento

Para comprobar que este modo de aprovisionamiento funciona, se carga el programa descrito en una ESP32, la cual, por tanto, actuará de aprovisionador. Por otro lado, se contará con otras dos placas que tienen implementados los modelos *Generic On/Off Server*.

Tan pronto como se inicializa la placa del aprovisionador, se empieza a hacer un escaneo de posibles dispositivos BLE dentro de su rango. En el momento en el que se enciende el primer nodo servidor, el nodo aprovisionador capta su mensaje de *advertising* y lo une a la red BLE Mesh, pues este pasa el proceso de aprovisionamiento y configuración de manera automática.

En la siguiente imagen se puede ver desde el monitor del dispositivo aprovisionador que pasos se van dando hasta completar este proceso.

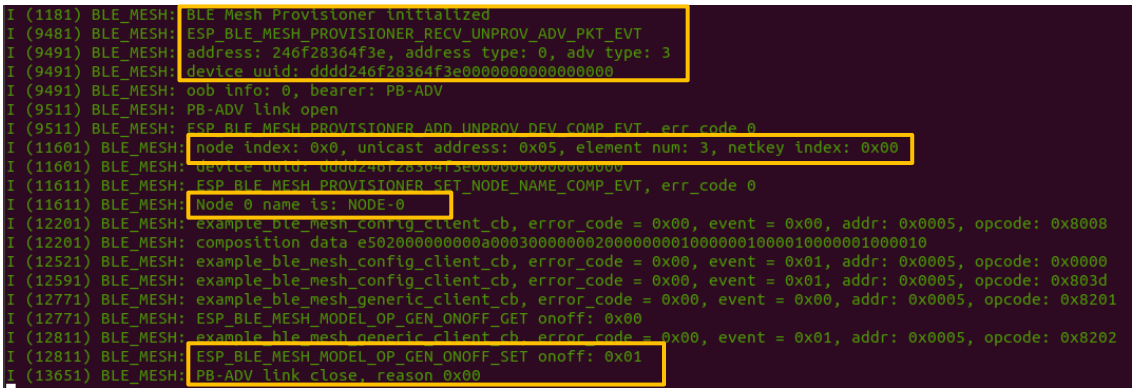

Figura 86. Proceso de aprovisionamiento del nodo 1 (monitor aprovisionador)

Entre estos pasos está la recepción de la MAC del nuevo nodo, la dirección *unicast* que se le ha asignado, el nombre del mismo, etc. Además, como último paso, se envía a través de un mensaje *Set* el cambio de estado *generic onoff*, pues lo que se pretende es que cuando el nuevo nodo esté incorporado a la red, este informe al usuario con una señal representada con el encendido de un LED.

Por su parte, en el monitor del primer dispositivo aprovisionado se tiene lo que se muestra en la Figura 87.

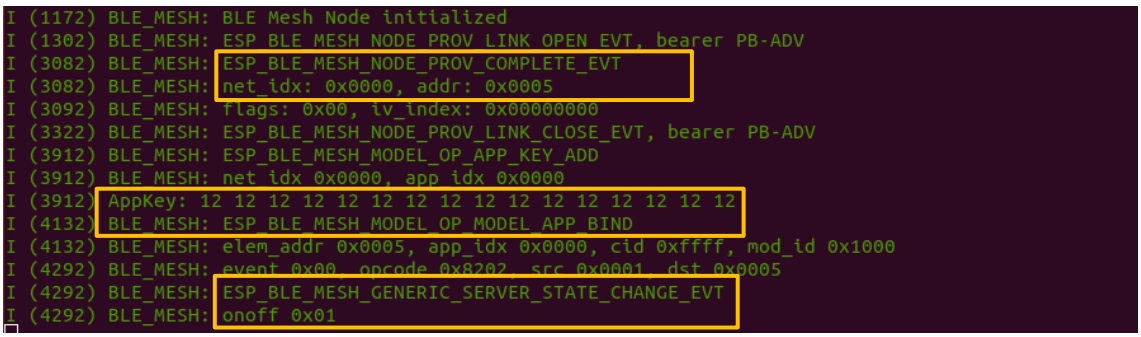

Figura 87. Proceso de aprovisionamiento de nodo 1 (monitor dispositivo aprovisionado)

Como se esperaba, en el nuevo nodo se confirma que se ha completado el proceso de aprovisionamiento a través del evento *ESP_BLE_MESH_NODE_PROV_COMPLETE_EVT*. El propio nodo conoce la dirección *unicast* que le ha sido asignada, y, además, se ha configurado correctamente la *AppKey*.

Finalmente, como se mencionaba antes, este recibe un mensaje por el cual debe cambiar el estado de su modelo *Generic On/Off Server* a encendido.

En este ejemplo, al no haber puesto límite al número de nodos que se pueden aprovisionar, se va a ver cómo sería el aprovisionamiento de un segundo nodo, teniendo en cuenta que los nodos que se quieran aprovisionar tras él lo harían de la misma manera.

Primero, en el monitor del dispositivo aprovisionador, se vería algo muy similar a lo que se mostraba en la Figura 86.

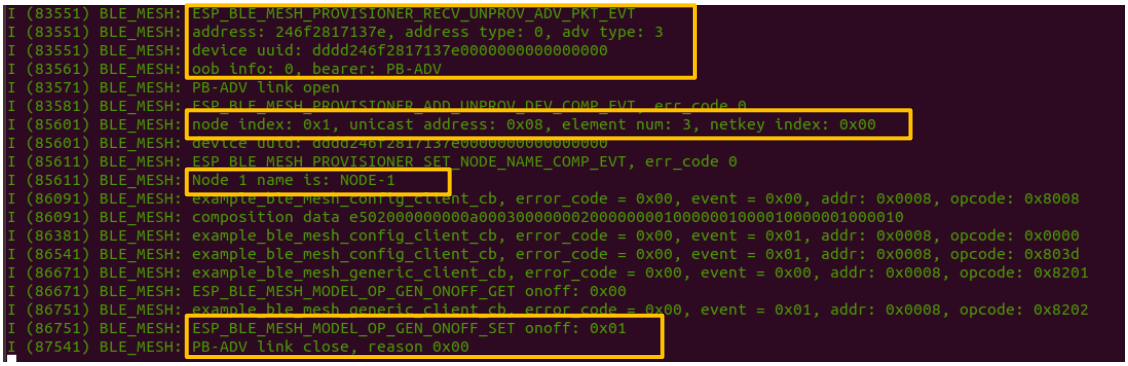

Figura 88. Proceso de aprovisionamiento del nodo 2 (monitor aprovisionador)

Obviamente lo que cambia es la dirección MAC del nuevo dispositivo que se quiere aprovisionar, la dirección *unicast* que se le asigna y el nombre del nuevo nodo. En este caso, se llama *NODE-1*. Por todo lo demás, se comporta igual.

En el monitor del nuevo dispositivo aprovisionado, se confirma, al igual que ocurría en el nodo anterior, que se ha completado el proceso de aprovisionamiento, a través del evento *ESP_BLE_MESH_NODE_PROV_COMPLETE_EVT*. Este segundo nodo conoce su dirección *unicast*, y, además, se configura correctamente la *AppKey*, que coincide con la del primer nodo. Finalmente, este nodo recibe un mensaje por el que el modelo *Generic On/Off Server* debe cambiar su estado a encendido, lo cual indicará al usuario a través de un LED que el nodo ha pasado el proceso de aprovisionamiento y configuración.

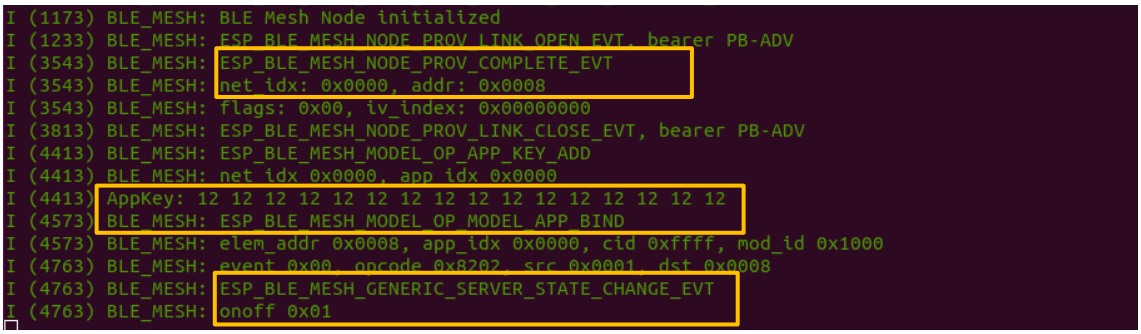

Figura 89. Proceso de aprovisionamiento de nodo 2 (monitor dispositivo aprovisionado)

B. Fast provisioning

Este modo de aprovisionamiento es una mezcla entre la solución de aprovisionar con una aplicación móvil y la solución de que un dispositivo BLE Mesh actúe como aprovisionador.

Cuando se trata de una gran cantidad de dispositivos no aprovisionados, por ejemplo, 100, aprovisionarlos uno por uno conlleva una gran cantidad de tiempo. Con un *fast provisioning*, el usuario puede ser capaz de aprovisionar estos 100 dispositivos BLE en 50 segundos aproximadamente.

ESP-IDF da soporte a este modo de aprovisionamiento, y en el directorio *esp-idf/examples/bluetooth/esp_ble_mesh/ble_mesh_fast_provision* de su GitHub oficial, se proporciona el código que es necesario *flashear* en los dispositivos que se quieran tener en una determinada red *mesh*. Asimismo, se proporciona la aplicación Android *EspBleMesh*[42], la cual fue desarrollada por la organización para este modo concreto de aprovisionamiento.

Obviamente, ilustrar en funcionamiento de *fast provisioning* a gran escala es complicado, puesto que no se cuenta con una gran cantidad de dispositivos. Aun así, a continuación, se desarrolla la demostración de cómo un conjunto de 4 nodos se aprovisionan de manera rápida, y como el usuario, a través de una aplicación (*EspBleMesh*), puede controlar todos a la vez o cualquiera de ellos de manera independiente.

Inicialmente, se cagar el programa *fast provisioning server* (*esp-idf/examples/bluetooth/esp_ble_mesh/ble_mesh_fast_provision/fast_prov_server/main*) en las cuatro placas que formarán la red BLE Mesh. En esta ocasión, no se hará un análisis detallado del código, ya que el objetivo aquí es estudiar cuál es el proceso del *fast provisioning*, sin embargo, es interesante saber que para lograr esto con éxito, cada nodo contará con un único elemento, en el cual irán implementados los siguientes modelos:

- *Configuration Server*: este modelo se utiliza para representar una configuración de red en malla de un dispositivo.
- *Configuration Client*: se usa para representar un elemento que puede controlar y monitorear la configuración de un nodo.
- *Generic On/Off Server*: es el modelo que implementa el estado de encendido o apagado del nodo.
- *Vendor Server*: este modelo implementa el estado *fast_prov_server* en el nodo.
- *Vendor Client*: es usado para controlar el estado *fast_prov_server*, que define el comportamiento de *fast provisioning* de un nodo.

[42] *EspBleMesh* App for Android. Link de descarga: http://download.espressif.com/BLE_MESH/BLE_Mesh_Tools/BLE_Mesh_App/EspBleMesh-0.9.4.apk

Por otra parte, como se verá más adelante, cuando los cuatro dispositivos pasan a formar parte de la red en malla, los nodos aprovisionados toman roles diferentes. Por un lado, el *smartphone* tomará el rol de *Top Provisioner*, mientras que el primer dispositivo que se aprovisione será *Primary Provisioner*. Por otro lado, el dispositivo que ha sido aprovisionado, pero cambia su rol a *Provisioner* se le llama *Temporary Provisioner*, sin embargo, si ese dispositivo no cambia de rol, mantiene el nombre de nodo.

Tal vez estos nuevos términos cuestan un poco de asimilar, por ello, al final de esta demostración se mostrará un diagrama de flujo (Figura 99) que resumirá de manera visual los pasos seguidos para conseguir un aprovisionamiento rápido.

Cuando se inicia la aplicación *EspBleMesh* (Figura 90), se deben seguir una serie de pasos para establecer una red BLE Mesh y controlar cualquiera de los nodos.

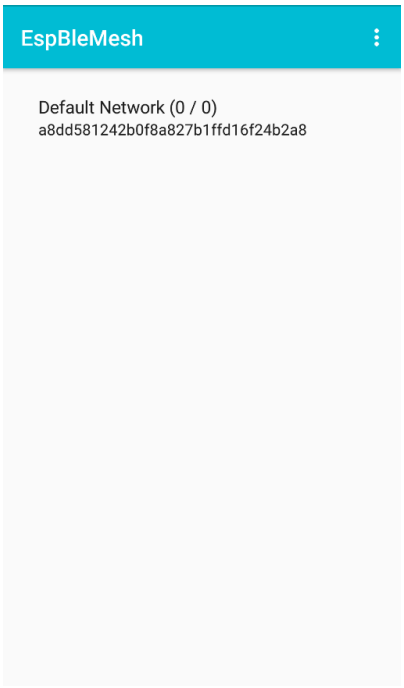

Figura 90. Inicio de la aplicación EspBleMesh

Tras pulsar los tres botones que aparecen en la esquina derecha de la figura 88, se selecciona *Provisioning* para escanear dispositivos no aprovisionados cercanos.

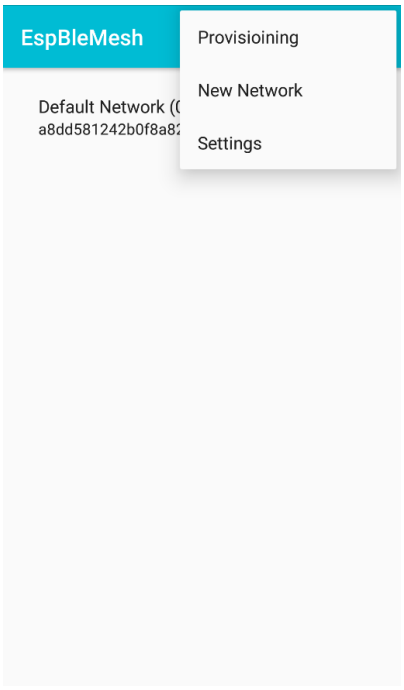

Figura 91. Provisioning

De los cuatro dispositivos con los que se cuenta, se selecciona uno de la lista escaneada, y, después, se ingresa la cantidad de dispositivos que se desea agregar a la red en malla que se está construyendo.

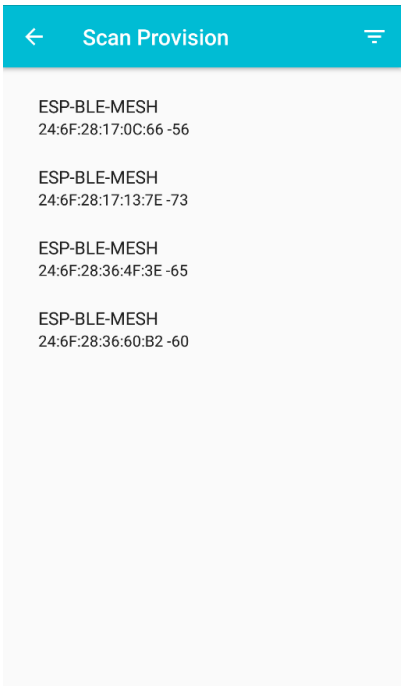

Figura 92. Lista de dispositivos no aprovisionados

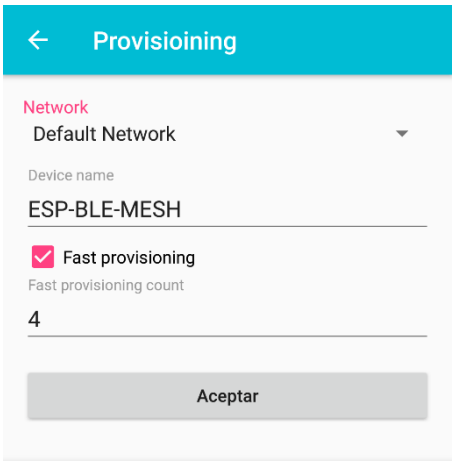

Figura 93. Número de dispositivos en la red BLE Mesh

En el monitor serie del dispositivo seleccionado, se puede ver como *Top Provisioner* lo configura para acceder con el portador GATT. Además, *Top Provisioner* envía el mensaje *send_config_appkey_add* para asignar la *AppKey* a este dispositivo.

```
I (1204) FAST_PROV_SERVER_DEMO: BLE Mesh Fast Prov Node initialized
I (430244) FAST_PROV_SERVER_DEMO: ESP_BLE_MESH_NODE_PROV_LINK_OPEN_EVT, bearer: PB-GATT
I (430954) FAST_PROV_SERVER_DEMO: ESP_BLE_MESH_NODE_PROV_LINK_CLOSE_EVT, bearer: PB-GATT
I (430954) FAST_PROV_SERVER_DEMO: ESP_BLE_MESH_NODE_PROV_COMPLETE_EVT
I (430954) FAST_PROV_SERVER_DEMO: net_idx: 0x0000, unicast_addr: 0x0002
I (430964) FAST_PROV_SERVER_DEMO: flags: 0x00, iv_index: 0x00000000
I (431614) FAST_PROV_SERVER_DEMO: example_ble_mesh_config_server_cb, event = 0x00, opcode = 0x0000,
addr: 0x0001
I (431614) FAST_PROV_SERVER_DEMO: Config Server get Config AppKey Add
```

Figura 94. Configuración del primer dispositivo

A continuación, *Top Provisioner* envía el mensaje *send_fast_prov_info_set* para proporcionar la información necesaria para que el dispositivo con dirección *unicast 0x0002* pueda cambiar su rol a *Primary Provisioner*. Además, como se puede ver en la siguiente figura, en el *Primary Provisioner* se dispara la API *esp_ble_mesh_set_fast_prov_action* a través del evento *ESP_BLE_MESH_SET_FAST_PROV_ACTION_COMP_EVT* para desconectarse del *Top Provisioner*.

```
I (432034) FAST_PROV_SERVER_DEMO: ESP_BLE_MESH_SET_FAST_PROV_INFO_COMP_EVT
I (432034) FAST_PROV_SERVER_DEMO: status_unicast: 0x00, status_net_idx: 0x00, status_match 0x00
I (432044) FAST_PROV_SERVER_DEMO: ESP_BLE_MESH_SET_FAST_PROV_ACTION_COMP_EVT, status_action 0x00
I (432064) FAST_PROV_SERVER_DEMO: ESP_BLE_MESH_MODEL_SEND_COMP_EVT, err_code 0
I (432064) FAST_PROV_SERVER: example_handle_fast_prov_status_send_comp_evt: opcode 0xc102e5
W (432074) FAST_PROV_SERVER: example_handle_fast_prov_status_send_comp_evt: Disable BLE Mesh Relay &
 GATT Proxy
```

Figura 95. Cambio de rol del dispositivo a Primary Provisioner

Tras poner en la aplicación (Figura 93) que los dispositivos que se quieren aprovisionar son cuatro, el *Primary Provisioner* envía el mensaje *send_config_appkey_add* a otro dispositivo para asignarle la *AppKey*. Después, el *Primary Provisioner* envía el mensaje *send_fast_prov_info_set* para proporcionarle la información necesaria para que el dispositivo pueda cambiar su rol a *Temporary Provisioner*. El dispositivo cuya MAC es *24:6F:28:17:13:7E*, llama a la API *esp_ble_mesh_set_fast_prov_action* para cambiarse así mismo al rol *Temporary Provisioner* y repite el proceso de aprovisionamiento con otro de los dispositivos.

Cabe destacar, que, con el fin de evitar conflictos a la hora de asignar direcciones, cada aprovisionador tendrá su propio rango de direcciones y un número máximo de nodos que puede aprovisionar. Finalmente, el *Temporary Provisioner* recopila las direcciones de los nodos que ha aprovisionado y las envía al *Primary Provisioner*.

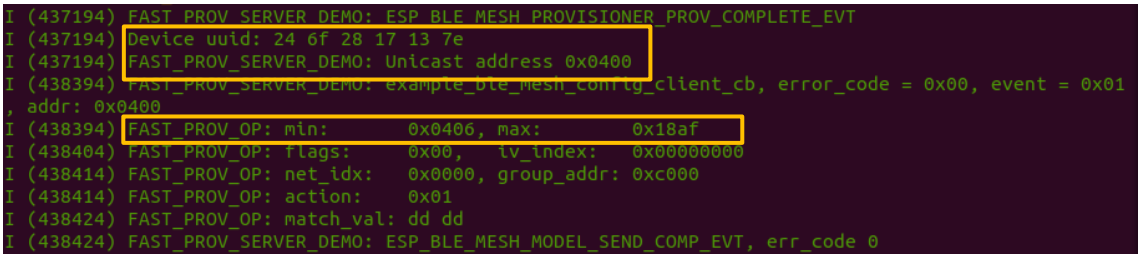

Figura 96. Temporary Provisioner (0x0400)

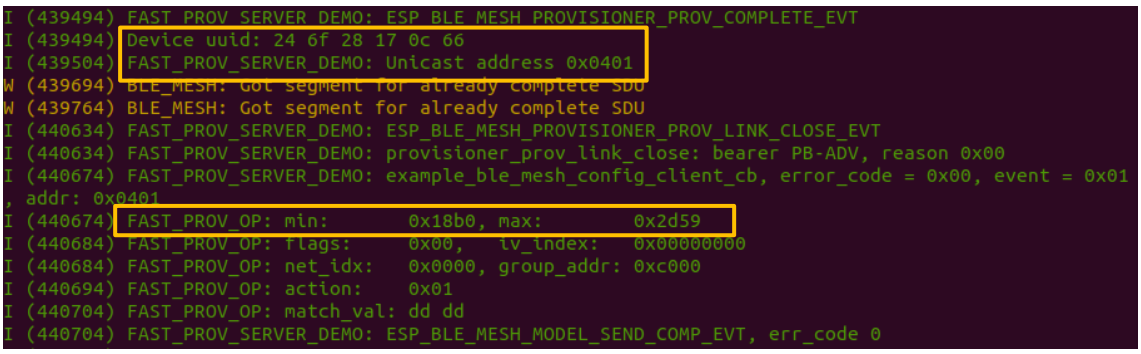

Figura 97. Temporary Provisioner (0x0401)

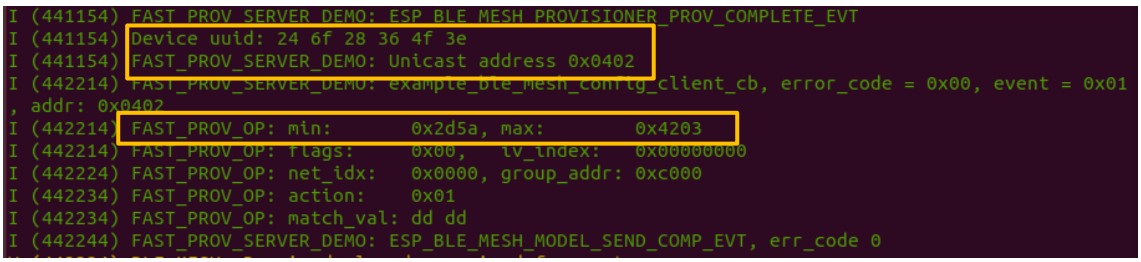

Figura 98. Temporary Provisioner (0x0402)

Una vez que los dispositivos estén aprovisionados, el *Primary Provisioner* se vuelve a conectar con el dispositivo móvil, es decir, con el *Top Provisioner*, el cual envía el mensaje *node_adress_Get* automáticamente después de conectarse con *Primary Provisioner*. En este punto, el *Top Provisioner* puede controlar cualquier nodo en la red Bluetooth Mesh.

Como se comentaba antes, tal vez este proceso, al incorporar nueva terminología, es un poco complejo de seguir, por ello, en la Figura 99 se muestra un diagrama de flujo que facilitará la compresión de esta demostración de *fast provisioning*.

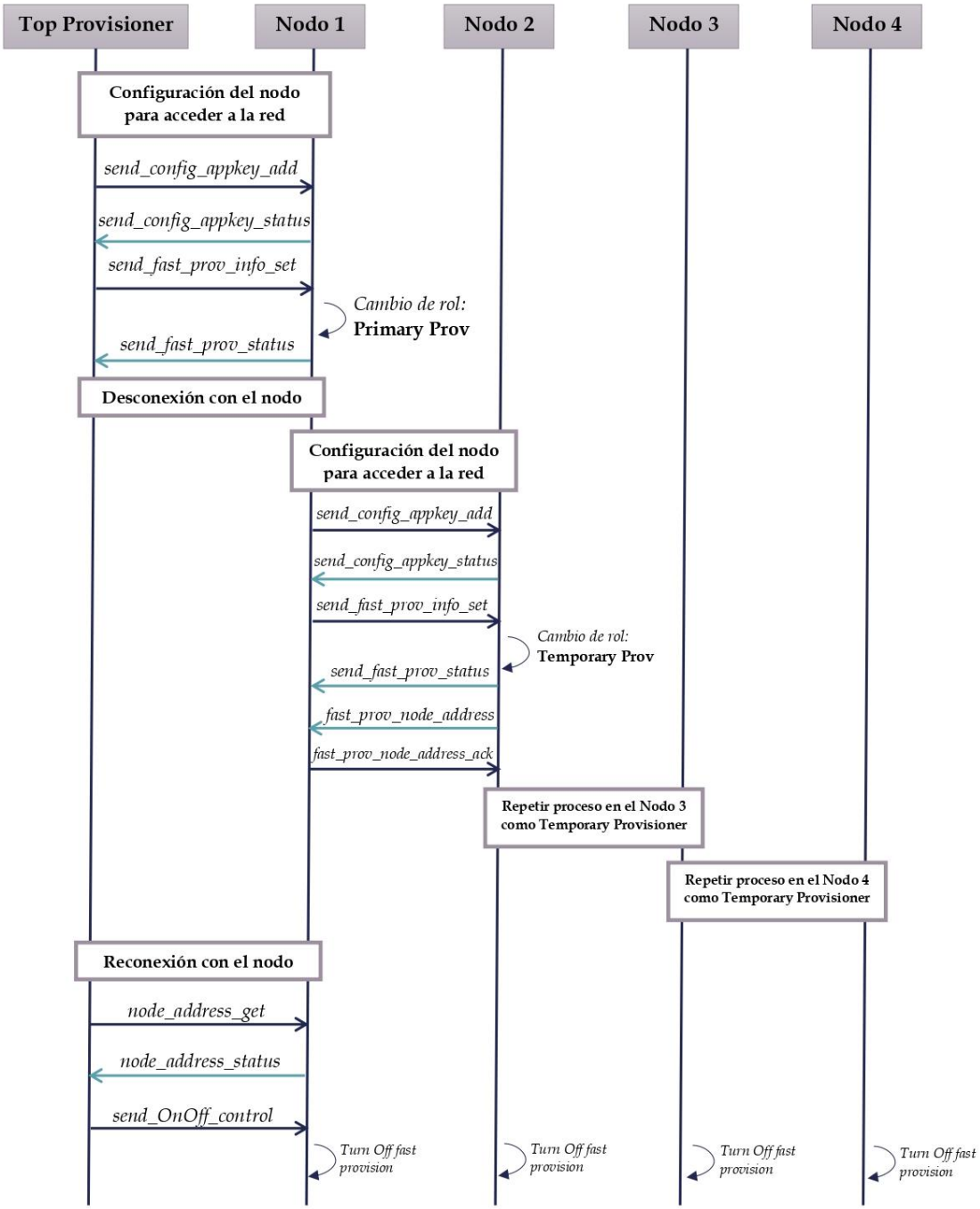

Figura 99. Diagrama de flujo de *Fast Provisioning*

Una vez creada y configurada la red BLE Mesh, el usuario puede controlar los diferentes nodos. En primer lugar, si se selecciona desde la propia aplicación *Fast Porvisioned* (Figura 100), se podrán ver los 4 dispositivos aprovisionados, y, desde la ventana de la Figura 101 se podrá controlar el encendido o apagado del led de cada una de las placas.

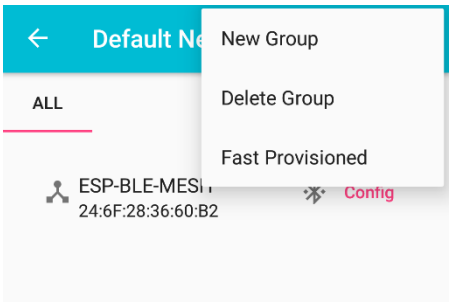

Figura 100. Fast Provisioned

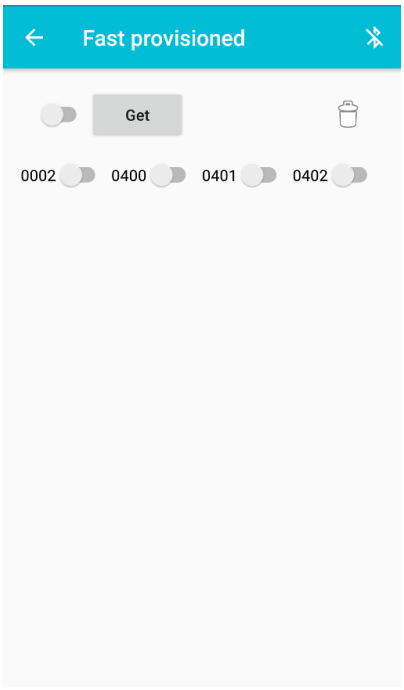

Figura 101. Control de los nodos

Al cambiar el estado del primer nodo, cuya dirección *unicast* es *0x0002*, en el monitor serie de dicho nodo se puede ver como el LED se enciende y el origen de este mensaje *Set* proviende del teléfono móvil, pues su dirección es *0x0001*.

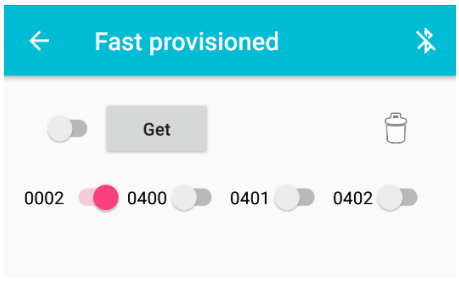

Figura 102. ON primer nodo

```
I (306564) FAST_PROV_SERVER_DEMO: event 0x00, opcode 0x8203, src 0x0001, dst 0x0002
I (306574) FAST_PROV_SERVER_DEMO: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (306584) FAST_PROV_SERVER_DEMO: onoff 0x01
W (306584) BOARD: led green is already on
```

Figura 103. Monitor serie del primer nodo

También se podría cambiar el estado del tercer nodo aprovisonado. En el monitor serie de este nodo se puede ver como el LED se enciende, puesto que desde el teléfono móvil se manda un mensaje *Set* a la dirección *0x0401*.

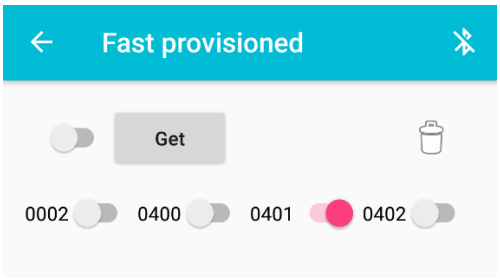

Figura 104. ON tercer nodo

```
I (978934) FAST_PROV_SERVER_DEMO: event 0x00, opcode 0x8203, src 0x0001, dst 0x0401
I (978934) FAST_PROV_SERVER_DEMO: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (978934) FAST_PROV_SERVER_DEMO: onoff 0x01
W (978944) BOARD: led green is already on
```

Figura 105. Monitor serie del tercer nodo

Sin embargo, si se quisiese cambiar el estado de todos los nodos a la vez, la dirección destino sería una dirección de grupo (*0xC000*) en la que se encuentran los cuatro nodos. Desde el monitor del primer nodo (*0x0002*) se vería como muestra la Figura 107.

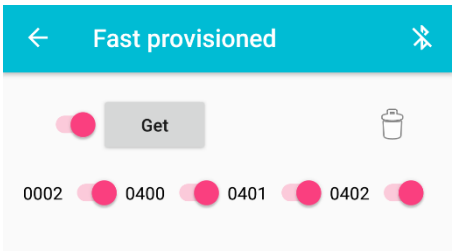

Figura 106. ON todos los nodos

```
I (1070914) FAST_PROV_SERVER_DEMO: event 0x00, opcode 0x8203, src 0x0001, dst 0xc000
I (1070924) FAST_PROV_SERVER_DEMO: ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT
I (1070934) FAST_PROV_SERVER_DEMO: onoff 0x01
W (1070934) BOARD: led green is already on
W (1070984) BLE_MESH: Replay: src 0x0001 dst 0xc000 seq 0x00015d
W (1071114) BLE_MESH: Replay: src 0x0001 dst 0xc000 seq 0x00015e
W (1071184) BLE_MESH: Replay: src 0x0001 dst 0xc000 seq 0x00015f
```

Figura 107. Monitor serie del primer nodo

C. BlueZ v5.50 en Raspberry Pi 3 como aprovisionador

En este último apartado, además de explicar una nueva forma de aprovisionamiento de nodos, se va a demostrar un requisito importante para cualquier protocolo de comunicación inalámbrica que admita una topología de red en malla. Este requisito es la interoperabilidad, pues como se verá más adelante distintos sistemas operativos, distintas arquitecturas y distintas implementaciones software, podrán interoperar vía Bluetooth Mesh.

BlueZ [43] es el proyecto oficial, continuamente en desarrollo, para la implementación de las especificaciones del estándar inalámbrico Bluetooth en sistemas operativos Linux. Esta implementación no solo proporciona un driver al kernel para poder hacer uso de dispositivos provistos de tecnología Bluetooth, sino que también proporciona drivers para las distintas formas de conectar un dispositivo Bluetooth al ordenador, por ejemplo, por USB, puerto serie, etc. Además, también cuenta con una serie de utilidades para la configuración de dispositivos. Obviamente, BlueZ apoya el perfil de malla de Bluetooth, pues a partir de su versión 5.47 proporciona la herramienta *meshctl* para configurar dispositivos de malla.

La versión con la que se va a trabajar de BlueZ será la 5.50, pues cuenta con algunas mejoras con respecto a las versiones anteriores. Además, con *meshctl* en esta nueva versión, será posible aprovisionar dispositivos de malla Bluetooth a través del portador GATT (PB-GATT) o el portador de publicidad (PB-ADV).

En esta ocasión, BlueZ v5.50 se implementa en la Raspberry Pi 3 para hacer de esta, gracias a la herramienta *meshctl*, un aprovisionador y configurador de productos BLE Mesh. Esto proporciona una nueva opción al lector para conocer las diversas formas de aprovisionamiento que permite esta novedosa tecnología inalámbrica.

A continuación, se debe seguir una guía paso a paso [65] en la que se muestra cómo configurar BlueZ en la Raspberry Pi 3. Si se requiere más información, Bluetooth SIG proporciona distintas guías de estudio (con distintas versiones de BlueZ) en la que se implementa BlueZ en una placa Raspberry Pi [67]. Sin embargo, la guía que se ha referenciado, es decir, la versión 1.2, es la que menos problemas genera. Esta guía lo que describe principalmente son los requisitos previos necesarios antes de comenzar, destacando que la tarjeta en la que está cargado el sistema operativo debe ser igual o superior a 16 GB. También describe cómo configurar la placa y cómo instalar las dependencias para BlueZ v5.50, además de enseñar a compilarlo e instalarlo. De igual modo, muestra como recompilar el *kernel* que crea la herramienta *meshctl*, para luego instalarlo recompilado y verificar la instalación de este.

En primer lugar, tras acceder de manera remota a la Raspberry Pi 3 y tener todo el entorno correctamente instalado, se puede abrir la herramienta *meshctl*, la cual se

[43] http://www.bluez.org/

encuentra en el directorio *~/blues-5.50/mesh*. En la siguiente figura, se pueden observar qué comandos, a través de un menú, ofrece.

```
pi@raspberrypi:~/bluez-5.50/mesh $ meshctl
[meshctl]# help
Menu main:
Available commands:
-------------------
config                                   Configuration Model Submenu
onoff                                    On/Off Model Submenu
list                                     List available controllers
show [ctrl]                              Controller information
select <ctrl>                            Select default controller
security [0(low)/1(medium)/2(high)]      Display or change provision security level
info [dev]                               Device information
connect [net_idx] [dst]                  Connect to mesh network or node on network
discover-unprovisioned <on/off>          Look for devices to provision
provision <uuid>                         Initiate provisioning
power <on/off>                           Set controller power
disconnect [dev]                         Disconnect device
mesh-info                                Mesh networkinfo (provisioner)
local-info                               Local mesh node info
menu <name>                              Select submenu
version                                  Display version
quit                                     Quit program
exit                                     Quit program
help                                     Display help about this program
export                                   Print evironment variables
[meshctl]#
```

Figura 108. Herramienta *meshctl*

Ahora es el momento de explicar cómo usar esta herramienta. Recordando, *meshctl* es una herramienta en BlueZ v5.50 que funciona como aprovisionador y distribuidor de los datos de aprovisionamiento (dirección *unicast*, *NetKeys*, IV index, etc.) a dispositivos nuevos no aprovisionados. Con ella, los usuarios pueden configurar los datos de aprovisionamiento por sí mismos a través de la edición del archivo *prov_db.json*.


```
{
  "$schema":"file:\/\/\/BlueZ\/Mesh\/schema\/mesh.jsonschema",
  "meshName":"BT Mesh",
  "IVindex":5,
  "IVupdate":0,
  "netKeys":[
    {
      "index":0,
      "keyRefresh":0,
      "key":"18eed9c2a56add85049ffc3c59ad0e12"
    }
  ],
  "appKeys":[
    {
      "index":0,
      "boundNetKey":0,
      "key":"4f68ad85d9f48ac8589df665b6b49b8a"
    },
    {
      "index":1,
      "boundNetKey":0,
      "key":"2aa2a6ded5a0798ceab5787ca3ae39fc"
    }
  ],
  "provisioners":[
    {
      "provisionerName":"BT Mesh Provisioner",
      "unicastAddress":"0077",
      "allocatedUnicastRange":[
        {
          "lowAddress":"0100",
          "highAddress":"7fff"
        }
      ]
    }
  ],
}
```


Figura 109. Archivo *prov_db.json*

Como se ve en la Figura 109, antes de proceder al aprovisionamiento se puede modificar el valor de *IV Index* deseado, o, por el contrario, se puede cambiar la *NetKey* a través del campo "*key*" o las *AppKeys*. De igual modo, se pueden modificar el inicio y fin de las direcciones *unicast* que el aprovisionador asignará a los futuros nodos que se unan a la red *mesh.*

Volviendo a la Figura 108, los comando con los que se va a trabajar para construir una red en malla Bluetooth son *discover-unprovisioned* <*on/off*>, que busca dispositivos para aprovisionar, *provision* <*uuid*>, que inicia el proceso de aprovisionamiento, y *disconnect* <*dev*>, que desconectará cualquier dispositivo de la red.

Para ver en acción a esta herramienta, es necesario contar con dispositivos capaces de soportar la topología de red *mesh*. Por ello, en primer lugar, se contará con un par de placa *ESP32-DevKitC V4,* y en ellas estará *flasheado* el mismo programa que se trabajó en la prueba de concepto de la sección 6.1 del capítulo 6, el cual proporciona ESP-IDF y está en el directorio *esp-idf/examples/bluetooth/esp_ble_mesh/ble_mesh_node/onoff_server/main* de este framework. Recordando, este implementaba el modelo *Generic On/Off Server.*

En segundo lugar, para demostrar la interoperabilidad de Bluetooth, se hace funcionar *Zephyr* sobre una arquitectura x86 usando el emulador QEMU, que actuará como otro nodo que la herramienta *meshctl* desde la Raspberry Pi deberá aprovisionar.

En el capítulo 4 ya se comentó que *Zephyr Project* podía hacer funcionar Bluetooth Mesh, ya que cubría todas las capas de su pila, usando una de las placas a las que daba soporte o usando QEMU con BlueZ. Como ya se vio, a pesar de que *Zephyr* da soporte para las ESP32, no fue posible demostrar que funcionase, por tanto, en esta ocasión, se hará funcionar Bluetooth Mesh en *Zephyr* desde un emulador QEMU [68].

En primer lugar, es importante tener la pila de protocolos Bluetooth de Linux, es decir, BlueZ, funcionando sobre la máquina virtual de Ubuntu sobre la que se está realizando este proceso. Para asegurar que todas las herramientas de BlueZ están incluidas en la distribución Linux o, por el contrario, se quiere construir desde cero BlueZ, se deben seguir los siguientes pasos:

```
git clone git://git.kernel.org/pub/scm/bluetooth/bluez.git
cd blues
./bootstrap-configure --disable-android --disable-midi
make
```

Tras hacer eso, se debe comprobar que herramientas como *btproxy* y *btmon* están incluidas, y, a continuación, se habilitan las funciones experimentales de BlueZ para poder acceder a su funcionalidad BLE más reciente. Para ello, se edita la primera

línea de inicio de ejecución del demonio del archivo */lib/systemd/system/bluetooth.service* como:

```
ExecStart=/usr/libexec/bluetooth/bluetoothd -E
```

Finalmente, se vuelve a cargar y reiniciar dicho demonio.

```
sudo systemctl daemon-reload
sudo systemctl restart bluetooth
```

Ahora, para hacer posible ejecutar aplicaciones Bluetooth en el emulador QEMU, habrá que exportar el controlador Bluetooth desde el sistema operativo *host* (Linux) al emulador usando algunas de las herramientas mencionadas anteriormente de BlueZ.

Primero, hay que asegurar que el controlador Bluetooth de la máquina virtual esté inactivo. Desde el terminal se puede hacer rápidamente ejecutando *sudo service bluetooth stop*. Después, se usa la herramienta *btproxy* para abrir el socket UNIX de escucha.

```
lidia@lidia-Ubuntu:~/bluez$ sudo tools/btproxy -u -i 0
[sudo] contraseña para lidia:
Listening on /tmp/bt-server-bredr
```

Figura 110. Escucha activada con *btproxy*

A continuación, desde *Zephyr* se elige la aplicación de muestra Bluetooth Mesh, la cual está ubicada en el directorio *~/zephyrproject/zephyr/simples/bluetoth/mesh*, y se ejecuta la aplicación en QEMU de la siguiente manera:

```
west build -b qemu_x86 .
west build -t run
```

```
lidia@lidia-Ubuntu:~/zephyrproject/zephyr/samples/bluetooth/mesh$ west build -b qemu_x86 .
-- west build: build configuration:
       source directory: /home/lidia/zephyrproject/zephyr/samples/bluetooth/mesh
       build directory: /home/lidia/zephyrproject/zephyr/samples/bluetooth/mesh/build
       BOARD: qemu_x86 (origin: CMakeCache.txt)
-- west build: building application
ninja: no work to do.
lidia@lidia-Ubuntu:~/zephyrproject/zephyr/samples/bluetooth/mesh$ west build -t run
-- west build: build configuration:
       source directory: /home/lidia/zephyrproject/zephyr/samples/bluetooth/mesh
       build directory: /home/lidia/zephyrproject/zephyr/samples/bluetooth/mesh/build
       BOARD: qemu_x86 (origin: CMakeCache.txt)
-- west build: running target run
[0/1] To exit from QEMU enter: 'CTRL+a, x'[QEMU] CPU: qemu32,+nx,+pae
SeaBIOS (version rel-1.12.1-0-ga5cab58-dirty-20200202_184651-8999447e980d-zephyr
)
Booting from ROM..Optimal CONFIG_X86_MMU_PAGE_POOL_PAGES 7
*** Booting Zephyr OS build v2.2.0-rc1-290-ge4479e2fecee  ***
Initializing...
[00:00:00.010,000] <inf> fs_nvs: 8 Sectors of 1024 bytes
[00:00:00.010,000] <inf> fs_nvs: alloc wra: 0, 3f0
[00:00:00.010,000] <inf> fs_nvs: data wra: 0, 0
Bluetooth initialized
[00:00:00.140,000] <inf> bt_hci_core: No ID address. App must call settings_load()
[00:00:00.170,000] <inf> bt_hci_core: Identity: 34:f3:9a:a3:5c:d6 (public)
[00:00:00.170,000] <inf> bt_hci_core: HCI: version 4.2 (0x08) revision 0x0100, manufacturer 0x0002
[00:00:00.170,000] <inf> bt_hci_core: LMP: version 4.2 (0x08) subver 0x0100
[00:00:00.180,000] <dbg> bt_mesh_settings.mesh_commit: sub[0].net_idx 0xffff
[00:00:00.180,000] <inf> bt_mesh_main: Device UUID: 00000000-0000-0000-0000-0000000~
Mesh initialized
```

Figura 111. Inicialización de la aplicación Bluetooth Mesh de Zephyr

Llegado a este punto, se realiza una conexión segura (Figura 112), pues se ha permitido que la aplicación acceda al controlador Bluetooth.

```
lidia@lidia-Ubuntu:~/bluez$ sudo tools/btproxy -u -i 0
Listening on /tmp/bt-server-bredr
Opening user channel for hci0
New client connected
```

Figura 112. Conexión realiza con éxito

Según *Zephyr*, el ejemplo denominado como *mesh* (*~/zephyrproject/zephyr/simples/bluetoth/mesh*) es capaz de demostrar la funcionalidad de Bluetooth Mesh, tanto si lo usa una placa con soporte Bluetooth LE o si se ejecuta en el emulador QEMU con BlueZ ejecutándose en el *host*. El ejemplo en cuestión cuenta con un único elemento, el cual implementa varios modelos de la malla estándar, entre ellos, *Configuration Server* y *Generic On/Off Server*.

Cuando se ejecuta el *host* en una computadora conectada a un controlador externo, es muy útil poder ver el registro completo de intercambios entre los dos. Por tanto, si se quiere hacer rastreo del HCI (*human computer interaction*), se usa la herramienta *btmon* que está incorporada en BlueZ.

```
lidia@lidia-Ubuntu:~/bluez$ sudo btmon
Bluetooth monitor ver 5.54
= Note: Linux version 5.4.0-42-generic (x86_64)                                  0.716095
= Note: Bluetooth subsystem version 2.22                                         0.716099
= New Index: 34:F3:9A:A3:5C:D6 (Primary,USB,hci0)                      [hci0] 0.716101
= Open Index: 34:F3:9A:A3:5C:D6                                        [hci0] 23.780858
= Index Info: 34:F3:9A:A3:5C:D6 (Intel Corp.)                          [hci0] 23.780867
@ USER Open: btproxy (privileged) version 2.22                {0x0001} [hci0] 23.780872
< HCI Command: Reset (0x03|0x0003) plen 0                          #1 [hci0] 24.220820
> HCI Event: Command Complete (0x0e) plen 4                        #2 [hci0] 24.232465
      Reset (0x03|0x0003) ncmd 2
        Status: Success (0x00)
< HCI Command: Read Local Supported Features (0x04|0x0003) plen 0  #3 [hci0] 24.240607
> HCI Event: Command Complete (0x0e) plen 12                       #4 [hci0] 24.242449
      Read Local Supported Features (0x04|0x0003) ncmd 1
        Status: Success (0x00)
        Features: 0xbf 0xfe 0x0f 0xfe 0xdb 0xff 0x7b 0x87
```

Figura 113. Ejemplo de uso de la herramienta *btmon*

Una vez que se tienen los tres dispositivos Bluetooth activos y dispuestos a unirse a una misma red BLE Mesh, es el momento de usar la herramienta *meshctl* desde la Raspberry Pi [66].

Primero, se descubren los dispositivos cercanos no aprovisionados escribiendo el comando *discover – unprovisioned on*.

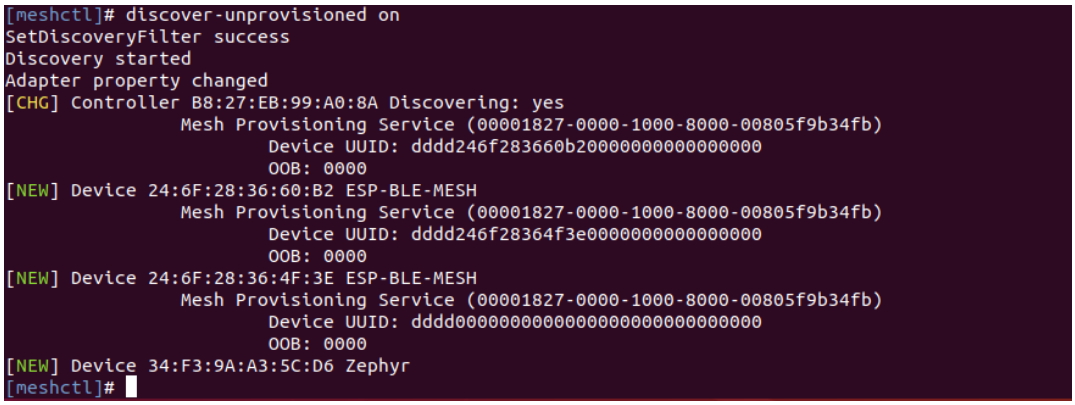

Figura 114. Descubrimiento de dispositivos.

Para proceder a la fase de aprovisionamiento es necesario quedarse con el UUID de los dispositivos. En primer lugar, se aprovisiona el dispositivo *Zephyr* a través del comando *provision* seguido de su UUID como se muestra en la Figura 115, pero después se debe hacer lo mismo con los otros dos dispositivos para que estos pertenezcan a la misma red BLE Mesh.

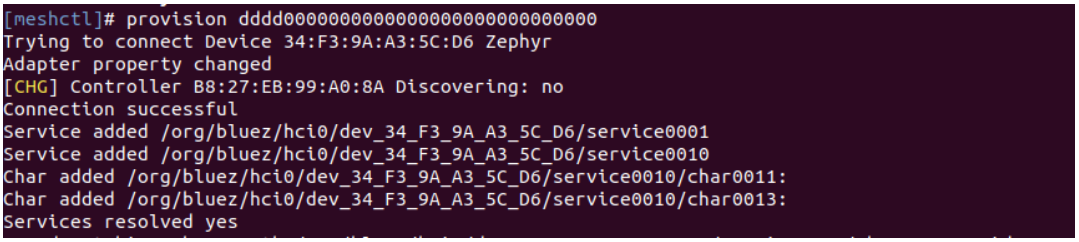

Figura 115. Aprovisionamiento del dispositivo de Zephyr

A continuación, *meshctl* solicita que se escriba el valor de salida OOB generado por el propio dispositivo.

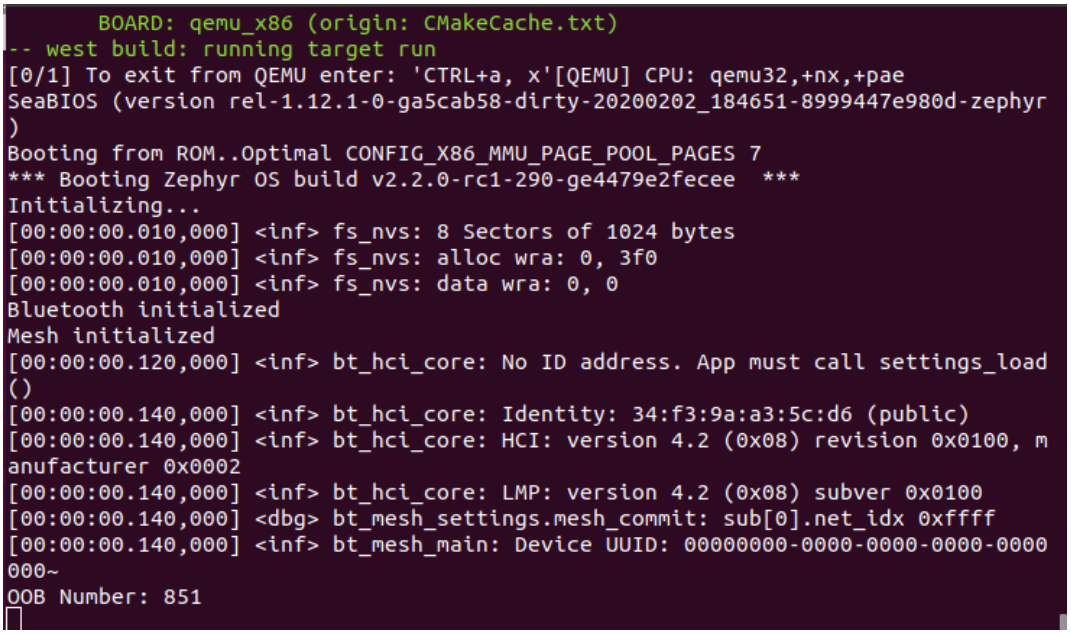

Figura 116. Monitor del dispositivo Zephyr

Figura 117. Monitor de la Raspberry Pi

Si el valor de OOB es correcto, *meshctl* avanza y llega al paso final (Figura 118), lo cual significa que el proceso de aprovisionamiento fue exitoso y el aprovisionador obtuvo los datos de composición del nodo recién aprovisionado.

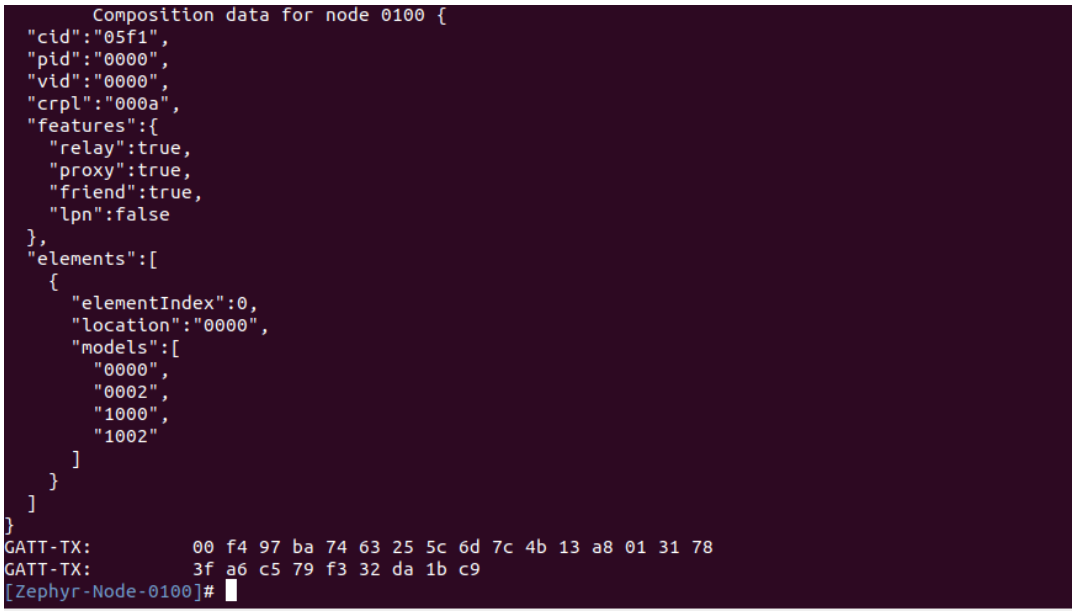

Figura 118. Datos de composición del nodo de Zephyr

Estos datos, además de ofrecer que el identificador de empresa del nodo (*cid*) es *0x05f1* y que sus características *relay*, *proxy* y *friend* están habilitadas, muestra que el nodo cuenta con un único elemento y este implementa cuatro modelos diferentes. Según el *id* de estos, los cuales vienen definidos en la especificación oficial de la malla Bluetooth [21], se determina que el *id 0x0000* pertenece al modelo *Configuration Server*, el *id 0x0002* al modelo *Health Server*¸ *0x1000* es el modelo *Generic On/Off Server* y el *id 0x1002* pertenece al modelo *Generic Level Server*.

A continuación, para que el aprovisionamiento quede completo, se realiza la configuración del modelo. Para ello, se usa el comando *menu config*.

```
[Zephyr-Node-0100]# menu config
Menu config:
Available commands:
-------------------
target <unicast>                                Set target node to configure
composition-get [page_num]                      Get composition data
netkey-add <net_idx>                            Add network key
netkey-del <net_idx>                            Delete network key
appkey-add <app_idx>                            Add application key
appkey-del <app_idx>                            Delete application key
bind <ele_idx> <app_idx> <mod_id> [cid]         Bind app key to a model
mod-appidx-get <ele_addr> <model id>            Get model app_idx
ttl-set <ttl>                                   Set default TTL
ttl-get                                         Get default TTL
pub-set <ele_addr> <pub_addr> <app_idx> <per (step|res)> <re-xmt (cnt|per)> <mod id> [cid]
                                                Set publication
pub-get <ele_addr> <model>                      Get publication
proxy-set <proxy>                               Set proxy state
proxy-get                                       Get proxy state
ident-set <net_idx> <state>                     Set node identity state
ident-get <net_idx>                             Get node identity state
beacon-set <state>                              Set node identity state
beacon-get                                      Get node beacon state
relay-set <relay> <rexmt count> <rexmt steps>   Set relay
relay-get                                       Get relay
hb-pub-set <pub_addr> <count> <period> <ttl> <features> <net_idx> Set heartbeat publish
hb-pub-get                                      Get heartbeat publish
hb-sub-set <src_addr> <dst_addr> <period>       Set heartbeat subscribe
hb-sub-get                                      Get heartbeat subscribe
sub-add <ele_addr> <sub_addr> <model id>        Add subscription
sub-get <ele_addr> <model id>                   Get subscription
node-reset                                      Reset a node and remove it from network
back                                            Return to main menu
version                                         Display version
quit                                            Quit program
exit                                            Quit program
help                                            Display help about this program
export                                          Print evironment variables
[Zephyr-Node-0100]#
```

Figura 119. Completar aprovisionamiento con *menu config*

Como se puede ver en la Figura 119, se despliega un submenú, desde el cual se pueden hacer operaciones de configuración y administración de la malla Bluetooth sobre un determinado elemento. El elemento primario de este nodo se identifica por la dirección *unicast 0x011b*, y para acceder a él se usa el comando *target 011b*. Una vez dentro, se le puede enlazar una *AppKey* a uno de los modelos que implementa a través de comando *bind*, puede suscribirse a una dirección de grupo, etc. Además, se puede verificar que el modelo *Generic On/Off Server* funciona, pues, a través del *menu onoff* se puede controlar el estado de los LEDs.

```
[Zephyr-Node-0101]# menu onoff
Menu onoff:
Available commands:
-------------------
target <unicast>                    Set node to configure
get                                 Get ON/OFF status
onoff <0/1>                         Send "SET ON/OFF" command
back                                Return to main menu
version                             Display version
quit                                Quit program
exit                                Quit program
help                                Display help about this program
export                              Print evironment variables
[Zephyr-Node-0101]# target 011b
Controlling ON/OFF for node 011b
[on/off: Target = 011b]# onoff 1
[on/off: Target = 011b]# onoff 0
```

Figura 120. Verificación del funcionamiento del modelo Generic On/Off Server